A64FX システムアプリ性能検討 WG

成果報告書

(WG 活動期間:2020 年 11 月-2022 年 10 月)

2022 年 10 月 31 日

サイエンティフィック・システム研究会

A64FX システムアプリ性能検討 WG

A64FX システムアプリ性能検討 WG 成果報告書　目次

1. はじめに
   - 1.1　当報告書について
   - 1.2　WG 活動概要
     - (1) 活動期間
     - (2) WG メンバー
     - (3) 活動実績

2. 高速化基盤
   - 2.1　ppOpen-AT における混合精度と消費電力を考慮した自動チューニング
     ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・ 片桐 孝洋 (名古屋大学)
   - 2.2　AI 計算の高速化
     ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・黒田 明義 (理化学研究所)
   - 2.3　低 B/F アーキテクチャ向けの三重対角行列解法
     ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・小野 謙二 (九州大学)

3. チューニング技法
   - 3.1　CFD 構造格子プログラムの PRIMEHPC FX1000 向け高速化チューニングについて
     ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・高木 亮治 (宇宙航空研究開発機構)
   - 3.2　CityLBM コードの A64FX 向け最適化および NVIDIA GPU・AMD GPU に対する性能調査
     ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・小野寺 直幸(日本原子力研究開発機構)
   - 3.3　ジャイロ運動論的シミュレーションコード GKV の性能分析および OpenMP TASK 構文による通信・演算オーバーラップの検討
     ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・前山 伸也 (名古屋大学)

4. アプリ高速化
   - 4.1　ボクセル有限要素法におけるニューラルネットワーク型前処理カーネルの SIMD 化に関する検討
     ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・藤田　航平 (東京大学)
   - 4.2　宇宙プラズマ２次元 Particle-In-Cell コード PIC2D の性能測定および推定
     ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・梅田 隆行 (名古屋大学)
   - 4.3　宇宙プラズマ５次元ブラソフコード Vlasov5D の性能測定および推定
     ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・梅田 隆行 (名古屋大学)
   - 4.4　FMO プログラム ABINIT-MP の高速化と超大規模系への対応
     ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・・望月 祐志 (立教大学)
   - 4.5　乱流燃焼コード LS-FLOW HO 性能改善検討
     ・・・・・・・・・・・・・・・・・・・・・・・・・・・・・熊畑 清 (宇宙航空研究開発機構)

別冊　C/C++プログラミングガイド　(富士通)

1 はじめに

1.1 当報告書について

スーパーコンピュータ「富岳」が本格運用され、現在、多種のスーパーコンピュータがメニーコア CPU を採用している。今まさに、メニーコア時代に突入している。しかしながらアプリ研究開発者にとっては、いまだに大規模コアの有効利用には様々な困難が伴う。その問題解決のためには、コンパイラ等のシステムソフトウェアと協調して性能最適化を行う知識と技術が利用者に求められるほか、システムソフトウェア自体の自動性能チューニングも必要である。そこで本 WG では、A64FX の ARM プロセッサ環境を中心に、コンパイラ、メッセージ通信ライブラリ、および、性能解析ツール等の改善点について議論し、そのノウハウの集約と共有を行った。

本 WG で取り扱った高速化基盤技術としては、混合精度計算と電力のチューニング、AI 計算の高速化、および、低 B/F アーキテクチャ向け数値解法に関する話題について議論した。またチューニング技法については、CFD 構造格子、格子ボルツマン法、および、ジャイロ運動論的シミュレーションコードの事例から、コンパイラの最適化性能や性能解析ツールの問題を検討・議論した。加えてアプリケーション高速化については、ボクセル有限要素法、Particle-In-Cell、ブラソフ、および、FMO のプログラムにおける高速化事例から、コンパイラ等の最適化性能を検討・議論した。本報告書では、上記の検討や議論の内容をまとめることに留まらず、別冊として C/C++プログラミングガイドの作成も行った。

以上の本 WG を通しての成果が、今後のスーパーコンピュータ活用時にユーザに益があるならば、望外の幸せである。

1.2 活動概要

(1)活動期間

2020年11月～2022年10月

(2)WG メンバー

			氏名	機関(2022 年 10 月 31 日現在)
会員	担当幹事		高木　亮治	宇宙航空研究開発機構
	推進委員	(まとめ役)	片桐　孝洋	名古屋大学
			熊畑　　清	宇宙航空研究開発機構
			小野　謙二	九州大学
			藤田　航平	東京大学　※会員外
			梅田　隆行	名古屋大学
			前山　伸也	名古屋大学
			小野寺直幸	日本原子力研究開発機構
			黒田　明義	理化学研究所
			望月　祐志	立教大学
賛助会員(富士通)	推進委員	(まとめ役)	井上　晃	富士通(株) (第 8 回会合まで)
			井上　俊介	富士通(株) (第 9 回会合から)
			野田　智之	富士通(株) (第 3 回会合まで)
			山中　栄次	富士通(株)
			鎌塚　俊	富士通(株)
			柴田　純	富士通(株)
			川島　崇裕	富士通(株)
			斉藤　盛幸	富士通(株)
			向井　優太	富士通(株)
			三浦　健一	富士通(株) (第 8 回会合まで)
			石井　邦憲	富士通(株)
			青木　正樹	富士通(株) (第 8 回会合まで)
			千葉　修一	富士通(株) (第 3 回会合まで)
			渡邊　健太	富士通(株) (第 4 回会合から第 8 回会合まで)
			松村　直樹	富士通(株) (第 4 回会合から第 8 回会合まで)
			原口　正寿	富士通(株) (第 4 回会合から)

			内海　裕一郎	富士通(株)（第4回会合から）
			成林　晃	富士通(株)（第4回会合から）
			渡辺　健介	富士通(株)（第4回会合から第8回会合まで）
			渡邉　雄二	富士通(株)（第4回会合から）
			稲荷　智英	富士通(株)（第9回会合から）
			高科　勝俊	富士通(株)（第9回会合から）
			笠井　良浩	富士通(株)（第9回会合から）
			野瀬　貴史	富士通(株)（第9回会合から）
			福本　尚人	富士通研究所
			川上　健太郎	富士通研究所（第3回会合から）

(3)活動実績

■第1回会合 ： 2020年11月27日(金)
- ・活動計画の検討（活動方針・活動内容・成果物など）
- ・今後の進め方の検討

■第2回会合 ： 2021年1月28日(木)
- ・WGで研究するアプリと課題の紹介（各参加メンバーより）
- ・A64FXでのAIに関する取組み紹介（富士通より情報提供）

■第3回会合 ： 2021年3月25日(木)
- ・各種アプリケーションの整理・仕分け
- ・熊畑先生発表(乱流燃焼コードLS-FLOW HO(LSHO)性能改善調査)
- ・梅田先生発表(宇宙プラズマの運動論シミュレーション)
- ・富士通研究所のAIについて検討

■第4回会合 ： 2021年7月12日(月)
- ・富士通側の体制変更について説明
- ・望月先生発表(FMOプログラムABINIT-MPの高速化と超大規模系への対応)
- ・前山先生発表(プラズマ乱流シミュレーションコードGKVの富岳での性能評価)

■第5回会合 ： 2021年9月15日(水)
- ・富士通A/I(GKV)調査結果説明
- ・富士通A/I(LS-FLOW)調査結果説明
- ・高木先生発表(CFDプログラムのPRIMEHPC FX1000向け高速化チューニングについて)
- ・藤田先生発表(A64FX環境における有限要素法地震シミュレーション)

■第6回会合 ： 2021年11月25日(木)
- ・小野寺先生(原子力CFDアプリケーションのA64FXへの移植・性能測定：
  JUPITER-ARMコードおよびCityLBMコードの最適化)
- ・小野先生発表(LSOR-PCR updates & PCR-like TPR)
- ・富士通A/I(TRIADカーネル性能)調査結果説明
- ・富士通A/I(高木先生のコード性能分析)調査結果説明

■第7回会合 ： 2022年2月24日(木)
- ・片桐先生発表(ppOpen-ATにおける混合精度演算と消費電力を考慮した自動チューニング)
- ・黒田先生発表(AI計算の高速化 昔のChainerとconvolution性能の評価)
- ・富士通A/I(藤田先生の質問(最内でないループへのSIMD適用))調査結果説明

■第8回会合 ： 2022年6月14日(火)
- ・高木先生発表(CFDプログラムのPRIMEHPC FX1000向け高速化チューニングについて(その2))
- ・梅田先生発表(プラズマ粒子コードの性能測定)
- ・富士通調査状況報告(CityLBMの調査)
- ・富士通調査状況報告(東大)藤田先生のリゾルバーの性能問題について)

■第9回会合 ： 2022年8月23日(火)
- ・富士通体制変更についての説明
- ・富士通調査状況報告(A64FX Gatherロード命令の性能について)

・富士通調査状況報告(ループ分割機能に関する AI)
・熊畑先生(乱流燃焼コード LS-FLOW HO(LSHO)性能改善調査)
・望月先生(FMO プログラム ABINIT-MP の高速化と超大規模系への対応(続報))
・小野寺先生(CityLBM コードの A64FX および GPU 環境での性能測定)
・報告書内容検討

■第 10 回会合 ： 2022 年 10 月 6 日(木)
・富士通調査状況報告(高木先生プログラムの性能分析結果)
・富士通「プログラミングガイド　入門編」についての説明
・成果報告書目次案の検討
・報告書作成状況確認

2.1 ppOpen-AT における混合精度演算と消費電力を考慮した自動チューニング

名古屋大学情報基盤センター　片桐孝洋

1. はじめに

ハードウェア性能の向上が見込めない時代―ポストムーア時代―が到来するといわれている。ポストムーア時代に向かう現在のエクサスケール時代でも、低電力化を指向したマルチコア化が進むと予想する。たとえばスーパーコンピュータ「富岳」では、1 ノードあたり 48 コア（4 ソケット）を搭載しており、ノード内の並列数が増加傾向にある。このようなノード内コア数増加の要因に加えて、低精度演算を活用して演算速度の高速化に寄与するハードウェア構成になる傾向がある。たとえば、AI 処理のための GPU では、AI に特化した高速な半精度演算器が搭載されていることは周知の事実である。またスーパーコンピュータ「富岳」でも、倍精度演算より約 4 倍高速な半精度演算器を搭載している。

以上の背景から、演算精度の劣化を考慮した上で、単精度演算や半精度演算を活用した**混合精度演算活用**による高速化が期待されている。一方で、混合精度演算を適用する対象のプログラム上の演算部分は多岐にわたる。そのため、**混合精度演算適用のためのプログラミングコスト増大**は避けられない。

そこで本報告では、混合精度演算を行う際のチューニングや速度向上と精度劣化の確認を含むプログラミングコスト削減を目指した、自動チューニング方式について紹介する。また同方式で消費電力を考慮したチューニングも可能となるため、簡単に紹介する。

2. 自動チューニング記述言語 ppOpen-AT

混合精度演算に限らず、性能チューニング処理を自動化できる計算機言語 ppOpen-AT を紹介する。ppOpen-AT[1]は、次世代の科学技術アプリケーションの開発・実行環境 ppOpen-HPC の自動チューニング機構として開発された自動チューニング(AT)言語である。並列数値計算ライブラリの開発効率を向上させることを目的に設計された FIBER 方式による AT 機能の付加を支援する AT 言語（ディレクティブ）である。

ppOpen-AT では、性能パラメタを設定しプログラムをチューニングする際に性能パラメタを変化させて性能を測定する枠組みを提供している[1]。実行時間を参照することで、多様な対象に対して AT が行える。ppOpen-AT により生成される最適化候補のうち、どの最適化候補コードを採用するかの選択を性能パラメタにする。本 AT 方式は、対象箇所の実行時間の計測結果から、どの候補を利用するかを決定する実行時最適化機能に関するものである。

3. 混合精度演算の AT 方式

混合精度演算の AT 方式では、（1）演算対象の計算精度を変化させる；（2）ユーザによりあらかじめ提示された演算誤差以下となる；という条件において、最高速となる候補

コードの組合せを選ぶ機能を提供する（この条件と手順については添付資料を参考)。

混合精度演算のプログラム最適化を目的にした ppOpen-AT において、プログラム上の対象は、①変数／配列、②ブロック、③関数／サブルーチンの 3 種類がある[2]。ここでは「②ブロック」の概要だけを説明する。この方式では

```
!oat$ MixedPrecision blocks, [clause. . . ]
{
!oat$ MixedPrecision block <1>
{structured-block}
!oat$ end MixedPrecision block <1>

!oat$ MixedPrecision block <2>
    {structured-block}
!oat$ end MixedPrecision block <2>
. . .
}
!oat$ end MixedPrecision blocks
```

のように記載する。

以上でユーザは、混合精度演算を適用したいプログラム上の箇所（ブロック）にディレクティブ“!oat$ MixedPrecision blocks”と“!oat$ end MixedPrecision blocks”を挿入することで、対象箇所を指定する。加えて、その対象箇所の内部を“!oat$ MixedPrecision block <num>”と“!oat$ end MixedPrecision block <num>”で指定する。この部分が、AT 対象の「ブロック」として定義される。ブロック単位で演算精度を変更することができる。ブロック番号は<num>で指定する。指定したブロック群をどのように低精度化するかは clause で指定する。演算精度の変更方法も、clause として指定する（詳細は文献[2]を参照)。

4. 予備評価結果

全球雲解像モデル NICAM（Nonhydrostatic ICosahedral Atmospheric Mode）を用いて予備評価を行った。NICAM のベンチマークパッケージの一つである nicam_dckernel_2016 の中で雲の微小な物理演算を行う physicskernel_microphysics のサブルーチン mod_mp_nsw6.f90 内の 3 重ループを対象プログラムとした。対象プログラムは一つの長い 3 重ループ構造で、ループ回数は固定である。ループ内で出力値を計算しており、ループ内の演算の低精度化は出力の計算精度に直接影響を与える。

ここで示すブロック方式の AT では、対象 3 重ループ内を計算のまとまりごとに 36 ブ

ロックに分割したうえで、ブロック単位で低精度化（倍精度⇒単精度）を行った。予備実験において測定する実行時間は、対象 3 重ループの実行時間である。演算精度は NICAM の出力 QV、QC、QR、QI、QS、QG の最大相対誤差として測定する。最大相対誤差の計算法などは、添付資料を参考のこと。

以上と同様の仕組みで全体の電力量が測定できる場合、電力最適化も実現可能である。本報告では掲載しないが、富士通社の PowerAPI を使うことで実装が可能である。詳細は文献[2]を参考されたい。

実験環境には、名古屋大学情報基盤センター設置のスーパーコンピュータ「不老」Type I サブシステム（Fujitsu PRIMEHPC FX1000）を使用した。

添付の実行結果から、オリジナルの All Block DP（全部倍精度演算）と比較して、各混合精度演算グループの速度向上が確認できる。速度向上率が最も大きいのは All Block SP（全部単精度演算）の場合で 1.56 倍である。また次点は、一部単精度演算となる混合精度演算のブロック 6 と 28 であり、それぞれ 1.12 倍の速度向上である。

興味深いことは、同様の速度向上率の 1.12 倍が得られるブロック 6 と 28 であるが、相対誤差は異なることである。この場合は、相対誤差の劣化がエンドユーザにより 2.90E-02 以下と指定される条件では、より精度の高いブロック 6 が最適である。つまり本提案手法により、速度向上率の上限、および**混合精度演算による精度劣化の度合いが低コスト（全自動）でチェックできる**点である。そのため、混合精度演算のチューニングツールとしても有効である例を示すことができた。

5. まとめ

本報告ではポストムーア時代に向けて高速な低精度演算を導入する傾向がある計算機構成のための自動チューニング（AT）の新機能の説明を行った。

既存の AT 言語 ppOpen-AT に実装されていない機能に対して予備実験を行った。NICAM を用いた予備実験の結果、②ブロック方式の AT で有効となる事例を示した。

本提案は混合精度演算での速度向上の目的に加えて、電力量削減の最適化も可能な AT 方式である。さらに、プログラム時の混合精度演算の有効性評価を完全自動化することで、高性能なプログラム開発の生産性も高めることが可能である。

スーパーコンピュータ「富岳」では、計算ノードが 48 コア+4 アシスタントコア構成となりノード内のコア数が従来の計算機より増加した。また、高速な単精度／半精度演算のハードウェア提供、さらに増大するキャッシュとメモリアクセス時間を考慮すると、低精度演算を活用した混合精度計算の活用がますます盛んになると予想される。また「富岳 NEXT」においても、アクセラレータ活用の方向もあるが、何らかの混合精度演算の活用は避けられないと著者は考えている。このような状況において本報告で示した AT 方式の適用とツール開発は、高性能ソフトウェアの生産性向上につながる技術の 1 つになるのではと、著者は考えている。

参考文献

[1] T. Katagiri and D. Takahashi, “Japanese Autotuning Research: Autotuning Languages and FFT”, Proc. of the IEEE, Vol. 106, Issue 11 (2018)

[2] 山梨祥平, 八代尚, 片桐孝洋, 永井亨, 大島聡史, “ppOpen-AT における演算精度と消費電力を考慮した自動チューニング方式の提案”, 情報処理学会研究報告, 2021-HPC-182, Vol. 12, pp. 1-9 (2021)

2.2 AI 計算の高速化

理化学研究所計算科学研究センター　黒田明義

1. はじめに

富岳は，開発プロジェクトである FS2020 終盤から，新規利用シーンとして AI 計算利用促進を意識した取り組みがなされた．これまでは GPU を用いて AI 計算を行なうというのが既成事実化されており，GPU での利用・開発が先行していた．これは AI の主要な処理である畳み込み積分計算（convolution）を単純かつ細かな GEMM 処理への置き換えが可能であったため，GPU のアクセラレータとしての特性を生かしやすかったためと言える．しかし GPU で出来る計算を汎用 CPU でできない理由はなく，理論的に GPU とチップあたりのピーク性能が 3TF と大きく変らない富岳 CPU でも高速な計算が可能と考えた．本調査では，汎用 CPU を用いた AI 計算で，性能問題に直面する例などをまとめた．

2. 富岳に向けての Chainer の高速化

2014 年からの FS2020 富岳開発プロジェクトで扱ってきた 9 本のターゲットアプリに加え，2018 年理研改組を機に，富岳の汎用 CPU 特性を生かした大規模 AI 計算の実現を目指して，当時の南ユニットにて調査並びにインポート，チューニングが行なわれた．

主に Chainer を用いて性能チューニングを試行し，「京」/ FX100 の CPU での AI 計算での実行・高速化の問題点を洗い出を行なった．専用の高速化ライブラリ（DNNL）を持たない「京」の CPU では，numpy の高速化が鍵であり，更に OpenBLAS を組み込むと高い性能が出ないため, SSL2 数値ライブラリの結合が必要であった. また optimizer 処理の 1/sqrt 計算にて SWPL がきかなかず，浮動小数点アンダーフローの対応が必要であり，これらの numpy 処理はまとめて Fortran に書き替えて高速化できた．Python によるスレッド非並列姓の問題は，MPI によるプロセス分割を行い高速化した．しかし分散並列を用いるとメモリ使用量が増えるため本手法は万能ではない．AI フレームワークによっては，コード内でスレッド番号を取得し，スレッド毎に明示的に処理を書き分ける内部実装で対応している．これらの性能チューニングを実施した結果「京」で 36.4 倍，ピーク比で 35.9%にまで高速化され，富岳でも AI 計算を十分できる見込みが立った．富岳では Resnet-50 性能で 66ips を達成し，ピーク性能比で Volta に匹敵する性能を実現した．現在，富士通研が開発した DNN 高速化ライブラリ(MKL DNN/oneDNN)を組み込んだ PyTorch / TensorFlow では 110ips を達しており，富岳並びに FX1000 で使用可能である．

3. numpy sqrt 処理

Chainer を使った MNIST では，フィルタ更新処理に相当する optimizer 内で行なわれる配列全要素の 1/sqrt 計算のコストが大きかった．そもそも「京」も富岳も sqrt や逆数計算は，級数展開したものを SWPL 処理で重ねるため，レイテンシが隠蔽され高速に計算でき

る．では何故 Chainer での 1/sqrt 計算は遅かったのか？Chainer 内の optimizer について Python の line profile を用いて取得した結果が以下の通りである．

```
Line# Hits Time         Per Hit % Time  Line Contents
==============================================================
104     7200 4482481.0    622.6   21.2    m += (1 - hp.beta1) * (grad - m)
105     7200 4366618.0    606.5   20.6    v += (1 - hp.beta2) * (grad * grad - v)
        7200 7337882.0   1019.2   34.6    param.data -= hp.eta * (self.lr * m /
                                            (numpy.sqrt(vhat) + hp.eps) +
113     7200 4864633.0    675.6   23.0    hp.weight_decay_rate * param.data)
```

コストが多い部分 sqrt などの配列要素の演算であり，Chainer では numpy が使用されていた．numpy は多くのアーキで使用可能で，一般に Python を用いて高速に数値計算が出来ると信じられている．しかしアーキごとに最適化されている関数でオーバーロードされないと以下の汎用ルーチンが動く．

```
core/src/umath/loops.c.src:
NPY_NO_EXPORT void
@TYPE@_sqrt(char **args, npy_intp *dimensions, npy_intp *steps, void *NPY_UNUSED(func))
{       if (!run_unary_simd_sqrt_@TYPE@(args, dimensions, steps)) {
            UNARY_LOOP {
                const @type@ in1 = *(@type@ *)ip1;
                *(@type@ *)op1 = @scalarf@(in1);
}       }       }
```

マクロを展開すると，以下のようなループ処理を行なっている．

```
build/src.linux-aarch64-3.8/numpy/core/src/umath/loops.c:
NPY_NO_EXPORT void
DOUBLE_sqrt(char **args, npy_intp *dimensions, npy_intp *steps, void *NPY_UNUSED(func))
{       if (!run_unary_simd_sqrt_DOUBLE(args, dimensions, steps)) {
            char *ip1 = args[0], *op1 = args[1];¥
            npy_intp is1 = steps[0], os1 = steps[1];¥
            npy_intp n = dimensions[0];¥
            npy_intp i;¥
            for(i = 0; i < n; i++, ip1 += is1, op1 += os1) {
                const npy_double in1 = *(npy_double *)ip1;
                *(npy_double *)op1 = npy_sqrt(in1);
}       }       }
```

numpy の最新バージョンでは関数名も抽象化されているため，sqrt などの数学関数が実装されている場所の判別も難しいので注意が必要である．このループは，入力と出力のポインタ ip1, op1 が重ならないことが分らないと演算の順序の入れ替えなどの最適化を行なうと結果異常になり，一般的に最適化は不可能といえる．実際 OpenMP でスレッド並列化すると結果が変った．

OCL 指示子の norecurrence を指定しても最適化は促進されない．これはインクリメント部が複数のポインタで構成されており，最適化の解析対象から除外されるためである．C++ で用いられる iterator も同様であるため注意を要する．例え sqrt 計算だけを SIMD 化し，SWPL 化できても，optimizer 処理全体ではパイプラインがぶつ切りになってしまう．このため numpy の最適化はあきらめ optimizer 処理全体を Fortran カーネルへの置き換え，高速化を実現した．富岳にも cupy のような数式全体を最適化する機能があると望ましい．

本 WG にて，一般的に用いられている数値計算ライブラリの処理内容について共有したところ，ip1 += is1 などのインクリメントをループボディに記載し，norecurence 指示子を付与してループ依存性を明示的に示せば，高速化するのではないかとの指摘を頂いた．しかし numpy は Python を構築時の FCC -Nclang で構築されており，clang モードでは，きめ細やかな SWPL 最適化が不可能であった．旧富士通研究所では，aarch64 向けに SVE 化した関数を作成し，オーバーロードして高速化する専用の numpy を開発中であり，近々提供される．

4. convolution 処理

AI では，離散畳み込み計算（convolution）が主要演算であり，以下のような数式であらわされる処理である．本処理は積和演算で構成される．

$$(f * g)(m) = \sum_{n} f(n)g(m-n)$$

f が入力関数で g をカーネル若しくはフィルタと呼ぶ．入力に対する出力応答の解析(電気回路，音響)・相関関数の計算(物理学)で使われる．物理学と AI とではフィルタ g の符号の向きが逆になることに注意が必要である．AI では，学習データ(画像データや音声データなど)からフィルタを介して，その特徴を抜き出す操作で用いる．Deep Learning ではこのフィルタリングを多数回実施し，それらのフィルタを最適化することが学習である．

ここでは，何故 6〜7 重ループ構造になるか？代替アルゴリズムの紹介し，「京」と GPU の性能比較，富岳での問題などを整理した．まず離散畳み込み演算をナイーブに書き下すと，以下の 3 重ループ構造になる．

- 最内ループ（内積の次元）：フィルタの 3 次元方向の要素数
- 中間ループ（内積の回数）：出力の 3 次元方向の要素数
- 最外ループ（まとめ処理の回数）

入力やフィルタは多次元であるために 6 重〜7 重ループにあることが多い．特に内側ループ

はフィルタサイズが小さい(1〜3 程度)ため，OCL SIMD＋ ループ collapse の効果を試したが，当時 OpenMP の SIMD が有効でなかったため，強制的に SIMD を効かせることができなかった．富岳では OpenMP SIMD がサポートされ使用できるが，今も強制 SIMD 化されたコードを生成せず，あくまでも SIMD 化を促す効果しかない．また多重ループを手動で collapse をすると，整数のアドレス計算が入るため性能が大幅に低下した．フィルタループをサイズ別に clone によりループ自動複製し，最適化促進を試みたが，同様に効果が得られなかった．

Intel コンパイラでは外側ループの SIMD を有効化させることが可能であり，高速化が可能であったが，富士通コンパイラでは，実現は難しいとの判断されている．gcc や llvm では富士通コンパイラ同様に外側ループの SIMD は実現出来ない上に，これらは OoO 最適化を期待し，SWPL 最適化レベルが低いため，現段階では富士通コンパイラを用いるのがベターであると言える．

convolution 処理は演算順序の入れ替えにより，処理を GEMM に置き換えることが可能である．またもともと畳み込み積分は物理学の世界では Fourier 変換を用いたり，実空間変換である Winograd 法による変換を行なうことで，演算数を減らす高速化ができるとされている．CPU ではこれらの汎用的な計算を得意とするため，これらの手法も有効であると考えられていた．当時のいくつかの CPU や GPU で様々なアルゴリズムでの convolution 計算の効率を比較したところ，「京」や ThunderX などの CPU では，Intel の MKL DNN，NADIA の cuDNN による性能とほぼ同等の効率を実現し得ることがわかった．しかし FFT では，2 巾など特徴的なサイズの convolution でしか高速化せず，Winograd 法も適用できる条件が厳しい．多くのケースで GEMM アルゴリズムが高速であったが，SIMD 長が増えたシステムでは，行列変換などの前後処理でインバランスが発生し効率があがらなかった．富岳の A64fx では，AI でよく使用される処理に関しては，Intel 向けの MKL DNN(後の oneDNN)に xbyak を用いて SVE 命令を自動生成し，高速化するライブラリを提供した．このライブラリを組み込むことができれば，GPU や Intel CPU に匹敵する効率を出し得る．

5. まとめ

本報告では，富岳上で AI 計算をする際に遭遇しえる性能に関する問題を提示した．AI 計算は convolution 計算とその他数値計算から構成されており，それらのライブラリ化並びに組み込みが高速化の鍵になる．

主要な処理である convolution 性能については，効率的には CPU でも GPU に劣らない性能を出し得ることが分った．cuDNN などの高速化ライブラリではどのアルゴリズムでも最終的に小さな GEMM の処理に帰着していた．但し，SIMD 長が大きくなってしまったシステムでは汎用的な方法（GEMM+前処理）だけでは高い効率は出ない．Python ではスレッド並列化が難しい問題もある．CPU では GPU 単体と比べてチップ性能が低い分，並列化でカバーする必要があるが，Python module と学習データの I/O・通信がネックになり，

並列性能が出にくい問題もある．精度要請からまとめ処理バッジサイズを大きくすることができず，CPU 向けにモデル並列が必要になる．

一般的な数値計算の例として，numpy sqrt の問題については，本来 sqrt 計算は，富岳は「京」ほどではないにせよ得意なはずである．富岳では，コンパイラがまだ製品のレベルに達していないという意見が多い．その多くは，「京」では性能が出たのに，富岳では出ないというものである．しかし富岳のコンパイラは「京」のコンパイラをアップグレードしたものであり，最適化レベルは落ちていない．性能が出ないのは，先に示した通り「京」と富岳でアーキテクチャが変ったためであることが多い．性能が出ない理由の多くは，SWPL による命令の多重実行ができないためである．その理由は，CPU 電力やチップ面積を削減するためにレジスタを削減したこと．それに対し市場性やユーザなどからの要望を優先して高周波数対応をしたため，その動作周波数を維持するために演算レイテンシが増加してしまったことが挙げられる．これらを補うべく富岳では OoO 資源が増やしたが，Intel CPU と比べると少ない．もともと「京」のコンパイラは SWPL による最適化を優先して作られていたため，OoO 資源を増やした分だけ汎用コンパイラでも性能が出やすくなったとは言えるが，他の CPU と比べると性能が見劣りするのはこのためである．富岳のコンパイラが未成熟でないと言っているわけではない．例えば，コンパイラに示唆を与えるタイプの指示子だけではなく，強制的に最適化を行なう指示子の提供や，外側 SIMD 化などチャレンジング方法で，性能向上の余地は多々ある．が，本体開発が終わった今，それらのアプリケーション開発環境の整備に工数が掛けられないのは大きな課題といえる．富士通が取り組む SWPL での高速化と llvm などでの OoO による高速化の良いとこ取りをした開発環境があれば良いが難しい．しかし一番の問題は，ユーザが OSS をブラックボックスとして使用していて，Intel CPU や gcc で高速で動くのだから富岳でも高速で動くはずであると思い込んでしまっていることであり，ユーザの意識改革や OSS の高速化をどう対応するかが重要な課題であると著者は考えている．

2.3 低 B/F アーキテクチャ向けの三重対角行列解法

九州大学情報基盤研究開発センター　小野謙二

1. はじめに

近年の計算機は動作周波数の上限に近づいた状況で演算性能を向上するため、演算器の多重化と共にキャッシュの多段化が基本的な構造となっているが、その副次的な作用としてメモリウォール問題を引き起こしている。特に、大型疎行列の係数行列をもつ連立一次方程式を解く場合に、その中核処理となる行列ベクトル積において、演算器の有効利用を図ることが難しい問題が顕わになっている。このような課題に対して、著者らは一次キャッシュの有効利用が可能な直接法と反復法のハイブリッド解法である LSOR-PCR 法を提案した[1]。この方法は、流体解析における圧力ポアソン方程式を差分法で離散化した場合などに得られる 7 点ステンシルの計算に利用できる。XY 方向は SOR などの反復解法を利用し緩和計算を行うが、最内側ループである Z 方向は三重対角行列を直接反転する。三重対角行列の解法として、ナイーブな LU 分解は SIMD 利用と並列化ができないため、PCR 法を基本とする。PCR 法は隣接方程式の係数の依存関係がなくなるように行列の基本操作を行い、互いに独立な 2 組の方程式群に分離する。この操作を縮約操作と呼ぶ。1 回の縮約により、N 元の連立一次方程式は 2 組の N/2 元の連立一次方程式となる。この縮約操作を logN-1 回繰り返し、最終段は解が直接得られるアルゴリズムである。PCR 法の利点として、依存性のないベクトル計算ができるため、SIMD 演算器が利用できる。更に、演算密度を高めることができ一次キャッシュの利用率も高まる。これにより、メニーコア環境での一次キャッシュの利用率が 90%を超え、スレッド並列のスケーラビリティは非常に高いことが確認された。しかしながら、最内側ループのベクトル長が長くなり、キャッシュサイズを超えるようになると一次キャッシュの有効利用ができなくなり、性能劣化が生じるという問題点が明らかとなった。

そこで本報告では、三重対角行列を M 分割して解く Tree Partitioning Reduction (TPR) 法[2]を導入することにより、キャッシュ利用の状況を改善する試みについて報告する。

2. TPR 法

TPR 法は連立一次方程式を M 分割して、分割された領域内で縮約操作を施す。このときの処理は M 並列で計算可能である。部分領域での縮約のため、方程式全体での縮約となるように整合性をとる必要がある。そのため、各領域から縮約した 2 つの方程式をマスターノード（あるいはスレッド）に集め、M 元の三重対角連立一次方程式を解く。このときの解法としては、Cyclic Reduction (CR)法を利用する。ここで得られた M 個の解は各領域にブロードキャストし、後退代入処理を行う。TPR 法の利点は、分割領域内だけで演算が可能で、データ参照の局所性が高い。したがって、L1 キャッシュの有効利用が期待でき

る。一方で、CR 法の短所である縮約段が増えるに従い並列度数が減少する、演算密度が低い、縮約の段数が多いという点を引き継いでいる。

3. PCR-like TPR (PTPR)法

オリジナルの TPR 法の短所を改善するため、TPR 法の CR 法に基づく縮約を PCR 法に変更する。これにより、段数が増えても並列度は低下せず、演算密度も高く保てる。また、アルゴリズム後半の後退代入を工夫することにより、わずか 3 ステップにでき、ステップ数を約半分にすることができる。詳細については[3]を参照。

4. ITO での数値実験

提案アルゴリズムを Skylake-SP Xeon Gold6154 および Tesla P100 で性能測定を実施した。実験手法は係数行列として Toeplitz 行列(-1/6,1,-1/6)、右辺(-1,1)を与え、$N=2^9$~2^{18}（GPU は 2^{16} まで)、M={1,2,4,…,32}とした。CPU での実行結果を図 1，2 に示す。

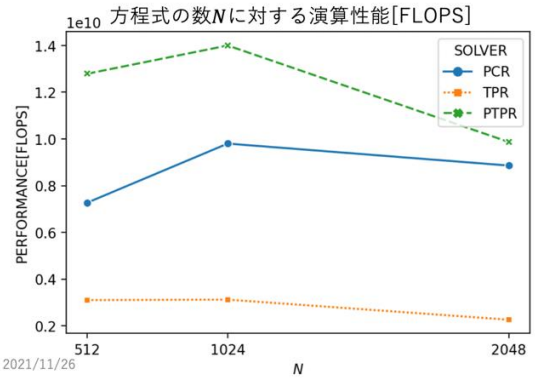


図 1　演算性能

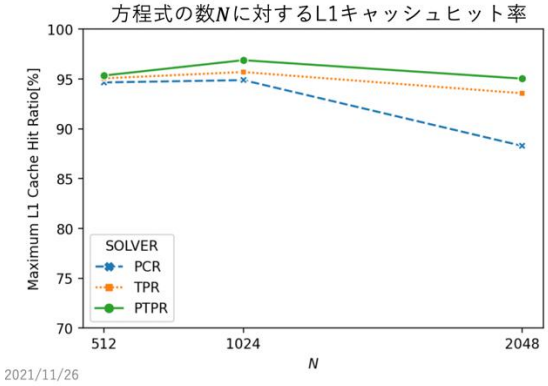


図 2 L1 キャッシュヒット率

期待するアルゴリズムの通り、N が大きくなると PCR 法のキャッシュヒット率が低下するのに対して、PTPR 法の性能劣化はわずかである。また、オリジナルの TPR 法よりも良いことがわかる。演算性能はループ内の演算密度が向上するため、TPR 法よりも大幅に良く、また、PCR を上回っている。図 3 には $N=2^{18}$ とした場合の強スケーリングの結果を示す。32 スレッド実行時に約 31 倍のスピードアップとなっている。分割の程度によって、キャッシュの利用状況が変わりスーパーリニアなケースも見られる。

GPU については、図 4 に実行性能を示す。PTPR 法は TPR 法に比べて約 6 倍の性能となっている。演算数は多くなっているが、ハードウェアを有効活用でき、より多くの処理が可能になっている。

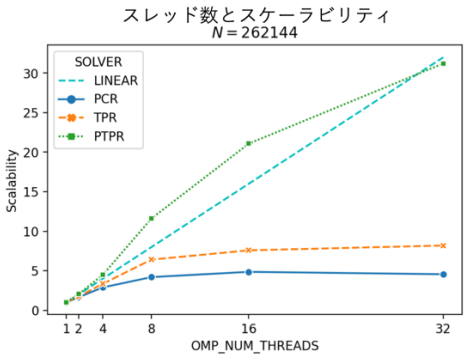


図 3　アルゴリズムによる性能比較

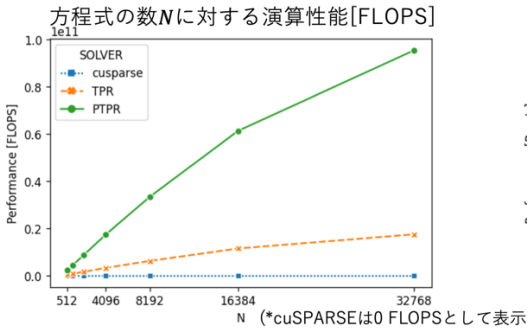


図 4　GPU の評価結果

5.　まとめ

本報告では、近年の低 B/F なメニーコア機をターゲットに、その性能を引き出せる疎行列解法アルゴリズムとして PCR-like TPR 法を提案し、その実行性能を Intel Skylake-SP で検証した。その結果、アルゴリズムの意図どおり L1 キャッシュのヒット率を改善でき、性能向上を図ることができた。パラメータとして最適な分割数があるがキャッシュサイズやコア数などのアーキテクチャ依存となり AT の余地がある。

参考文献

[1] K. Ono, T. Kato, S. Ohshima, and T. Nanri. 2020. Scalable Direct-Iterative Hybrid Solver for Sparse Matrices on Multi-Core and Vector Architectures. In Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region (HPCAsia2020). Association for Computing Machinery, New York, NY, USA, 11–21. https://doi.org/10.1145/3368474.3368484.

[2] A. P. Dieguez, M. Amor, and R. Doallo, “Tree partitioning reduction: A new parallel partition method for solving tridiagonal systems,” ACM Trans. Math. Software, vol. 45, 3, pp. 1–26, August 2019. DOI:https: //doi.org/10.1145/3328731.

[3] T. Mitsuda and K. Ono, "A Scalable Parallel Partition Tridiagonal Solver for Many-Core and Low B/F Processors," 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2022, pp. 860-869, doi: 10.1109/IPDPSW55747.2022.00142.

3.1 CFD 構造格子プログラムの PRIMEHPC FX1000 向け高速化チューニングについて

宇宙航空研究開発機構 宇宙科学研究所　高木亮治

1. はじめに

FUJITSU Supercomputer PRIMEHPC FX1000[1)]（以後 FX1000 と呼称）は FUJITSU Supercomputer PRIMEHPC FX100（以後 FX100 と呼称）に比べて、CPU に関しては SIMD（Single Instruction Multiple Data）長およびコア数の増加、および HBM2（High Bandwidth Memory 2）の採用によるメモリアクセス性能の向上などによる演算性能の向上が図られている。その一方で、最適化を実施する上で重要となるレジスタ数が減少しており、これまで FX100 向けに行っていた高速化チューニングの手法を大きく見直す必要がある。ここでは、構造格子を用いた CFD ソルバーを対象とした FX1000 向け、特に CPU 単体性能の高速化チューニング手法について紹介する。

2. PRIMEHPC FX1000

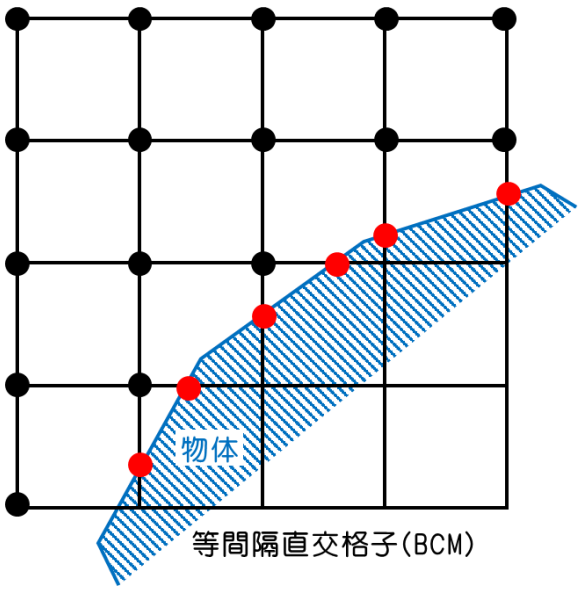
FX1000 はスーパーコンピュータ「富岳」の商用機と位置付けられるマシンである。FX1000 の CPU である A64FX は Arm8.2-A 命令セットアーキテクチャを HPC（High Performance Computing）向けに拡張した SVE（Scalable Vector Engine）を実装したプロセッサで、48 個の演算コアと 2 個のアシスタントコアを持ち、周波数が 2.2GHz で理論演算性能は 3.3792TFLOPS となる。メモリに関しては、HBM2 を 4 セット 32GByte 搭載することで、1.024GB/s の比較的高いメモリバンド幅を有する。各計算ノードではこの A64FX を 1 個搭載し、ノード間の接続には Tofu インターコネクト D が用いられている。

3. FX1000 向け高速化チューニング手法

高速化チューニングの観点で、FX100 と FX1000 を比較すると、まず CPU のアーキテクチャが SPARC から ARM に変更されたが、命令セットの違いに関してはコンパイラが対応するレベルのものであり、一般的にはユーザーは重視しなくても良いはずである。FX100 から FX1000 へは演算性能、メモリ性能が強化されているが、計算性能に大きく関係する違いとしては、コア数および SIMD 長の増加とレジスタ数の減少である。特に最適化の観点で見たときに一番大きな影響があるのがレジスタ数の減少である。レジスタ数の問題は先代の FX100 の頃から少しずつ顕在化しつつあった。FX100 における最適化戦略としては、最大性能を実現するためには、対象となるループに対して、スレッド並列、ソフトウェアパイプライン、SIMD を同時に適用する必要があった。更に、FX100 で使われているメモリである HMC の特性に起因して、XFILL と呼ばれるメモリアクセス量を削減する機能も併せて適用することが必要であった[2,3)]。しかしながら、これら全ての最適化機能を同時に適用するためには、我々が普段実装しているループ、例えば、対流項流束および粘性項流束の計算ループでは、ループボディが大き過ぎて、上記全ての最適化を同時に

適用しようとすると、レジスタが不足し全ては適用できない状態であった。FX100 から大幅にレジスタ数が削減された FX1000 においてはこの問題が顕著に発生した。これまでの一般的な圧縮性流体の解析プログラムの実装ではループボディが大きい、つまりループの中で実行する演算数が多すぎるので、ループを分割することが求められる。そのため、FX1000 ではコンパイラの機能として自動的にループを分割する機能や、ユーザーが分割する場所をコンパイラディレクティブで指定する機能を実現している。しかしながら、もともと 1 個のループを分割すると、分割されたループ間でデータの受け渡しが必要となり、分割前に比較するとメモリアクセスが増加する事になる。CFD の解析プログラムは往々にしてメモリアクセスが比較的多い傾向があるため、ループ分割はその傾向に拍車をかける。事実、単純にループ分割を行うと、分割したループに対しては、ソフトウェアパイプラインや SIMD など最適化が適用できるようになり、高速に実行できるようになるが、メモリアクセスが増加することで、メモリアクセスの待ち時間が増大し、結果的に分割前より遅くなることが多く見られた。

FX1000 における高速化チューニングとしては、レジスタ数の減少への対応策としてループ分割を如何に活用するかが重要となる。ここでは構造格子を用いた CFD プログラムを対象として、FX1000 向けの高速化チューニングを試みた。対象とした CFD プログラムは、現在開発を進めている階層型等間隔直交構造格子法と埋め込み境界法を用いたプログラム[4]である。マルチブロック構造格子法などの一般的な物体適合格子を用いた場合とは異なり、直交格子ではメトリックを扱う必要がないが、図 1a で示す様に直交格子の内部に物体が存在するため、その物体を表現する処理が必要となる。



a）壁の処理（●：流体、●：壁）

```
do k=istart,iend
do j=istart,iend
do i=istart,iend

  流体の計算（●）

  if( 壁がある（●）) then
    壁の処理
  endif

enddo
enddo
enddo
```

b）ループ構造の例

図 1 等間隔直交格子＋埋め込み境界法のプログラム例

一般的なやり方としては、前もって物体（壁）がある場所を示すフラグ（例；流体なら 0、物体なら 1）を用意し、フラグを if 文で判断しながら流体の処理、物体の処理を実施することになる。そのため、図 1b で示す様に、ループ中に if 文が存在することになり、最適化の適用はそのままでは困難となる。

一般的にループ中に if 文がある場合は、①マスク処理もしくは②if 文を分離することで最適化を適用することになる。

①マスク処理：

if 文の真偽に応じたマスク変数（前述したフラグがほぼ等価）を用意し、計算は if 文の真と偽の両方の計算を実施し、それらの結果にマスク変数をかけることで最終的な結果を得る手法である。if 文の真偽両方の計算を実施するので演算量が増えるが、if 文の真偽率で演算量（性能）がブロック間で変化しない、つまり全てのブロックで演算量が等しいという長所がある。

②if 文の分離：

if 文の真と偽でループを分割する。ここでの例では、まず計算領域内に物体がなく、全て流体として計算を行う。その後、前もって用意した壁がある部分のリストを用いて、壁の部分だけの計算を実施する。壁のある場合は、3 次元で各軸方向に対して、＋方向、－方向、両側の 3 パターンが存在し、それぞれを分けるので全部で 9 個となる。この手法は、元々の実装に対してほぼ演算数は増えないが、if 文の真偽率に応じて演算数（性能）がブロック間で変化するためブロック間の演算負荷バランスの観点では不利である。また、リストアクセスが発生するので性能が低下するといったデメリットがある。ここでは、対流項の計算には①マスク処理、粘性項の計算には②if 文の分離を適用することで、それぞれの計算ループ内での if 文を削除した。これらより、階層型等間隔直交構造格子法と埋め込み境界法を用いたプログラムに特有の if 文を含んだループを、ループ内に if 文がない多重ループとすることができる。このループ構造はマルチブロック構造格子など一般的な物体適合の構造格子と同じであり、以後紹介する手法は一般の構造格子を用いた CFD プログラムに対しても汎用的に適用可能なものである。

構造格子では、境界条件などを除き主要部分は多重ループ構造となるため、この多重ループの高速化チューニングについて述べる。FX1000 の特性を考慮した場合、高速化の戦略としては

(0) ループに対して、スレッド並列、ソフトウェアパイプライン、SIMD の適用

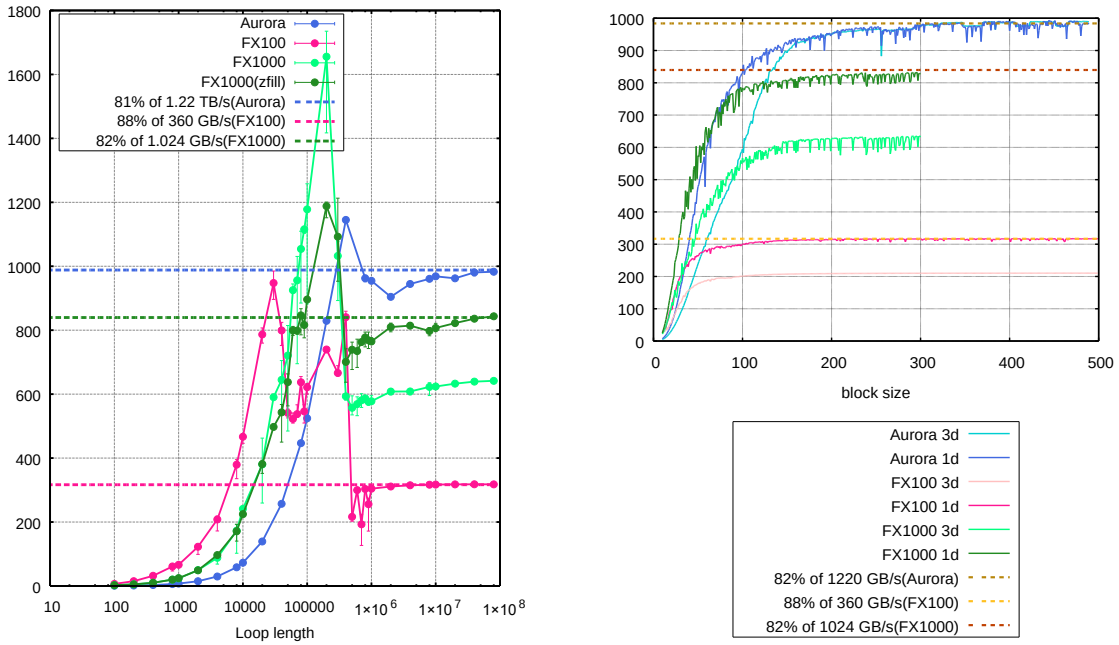
(1) メモリアクセス性能を引き出すために、連続アクセスかつループ長の確保

(2) レジスタ数に比べてループボディが大きいので、ループ分割の適用

(3) ループ分割に伴うメモリアクセスの増大を避けるためにブロッキングの適用

を行う。前述したループ内の if 文の削除などは(0)に該当するが、主要ループに対して、OpenMP もしくは自動並列によるスレッド並列をはじめ、ソフトウェアパイプライン、SIMD の適用阻害要因をなくし、これらの最適化が適用できるようにする。

(1)に関しては FX100 と同様にメモリ性能を引き出すためには連続アクセスと長いループ長が必要となる [2,3]。図 2a に STREAM TRIAD による FX1000 のメモリアクセス性能の実測値を示す。FX1000 では ZFILL（前述の XFILL と同等の機能）の利用で理論性能の82%程度の性能が出せている。なお、ZFILL の効果によりメモリ性能は向上しているが、その分 L2 キャッシュの性能は低下している。図 2b において「3d」は 3 重ループの結果、「1d」はループを 1 重化し長いループ長を確保した結果である。3 重ループでは、ループ長がブロックサイズ（横軸）のループが 3 重になっている。一方、1 重化したものは(ブロックサイズ)^{3}がループ長となる。この結果より、メモリアクセスの最大性能を出すためには、ある程度のループ長が必要なことがわかる。比較的長いループ長を実現するための手段として、ここではデータ配列の 1 次元化を用いている。



a）実測値の比較（横軸はループ長、縦軸はメモリアクセス性能[GB/s]を示す）　　b）1 重化の効果（横軸はブロックサイズ、縦軸はメモリアクセス性能[GB/s]を示す）

図 2 STREAM TRIADの性能

(2)に関してはコンパイラによる自動ループ分割を利用する。具体的には分割するループに対してコンパイラディレクティブで分割を指示する。(3)に関しては、ループ分割にともなって発生するループ間のデータの受け渡しがメモリアクセスではなく、L2 キャッシュ等で受け渡しができるように分割したループ長を制御する。図 3a で示す多重ループ（3 次元版の TRIAD）に対して上記の(1)～(3)を適用すると図 3b で示す様な実装になる。

```
real(8), dimension(:,:,:) :: a3d,b3d,c3d
```

```
real(8), dimension(:,:,:) :: a3d,b3d,c3d
```

```
!$omp parallel
!$omp do collapse(2)
do k=1,N
do j=1,N
do i=1,N
   a(i,j,k) = b(i,j,k) = S * c(i,j,k)
enddo; enddo; enddo
!$omp end do
!$omp end parallel
```

a）チューニング前

```
call kernel(a3d,b3d,c3d)

contains
   subroutine kernel(a1d,b1d,c1d)
      real(8), dimension(*) :: a1d,b1d,c1d
      !$omp parallel
      !$omp do
      do lb=lbsrt,lbend,lbsize   ←ブロックのループ
         !ocl loop_fission_target(LS)
         !ocl loop_fission_threshold(50)
         do l=lb,lb+lbsize-1      ←ブロックの計算
            a1d(l) = b1d(l) + S * c1d(l)
         enddo
      enddo
      !$omp end do
      !$omp do
      !ocl loop_fission_target(LS)
      !ocl loop_fission_threshold(50)
      do l=lbsrt_rest,lbend_rest   ←ブロックの余り
         a1d(l) = b1d(l) + S * c1d(l)
      enddo
      !$omp end do
      !$end parallel
end subroutine kernel
```

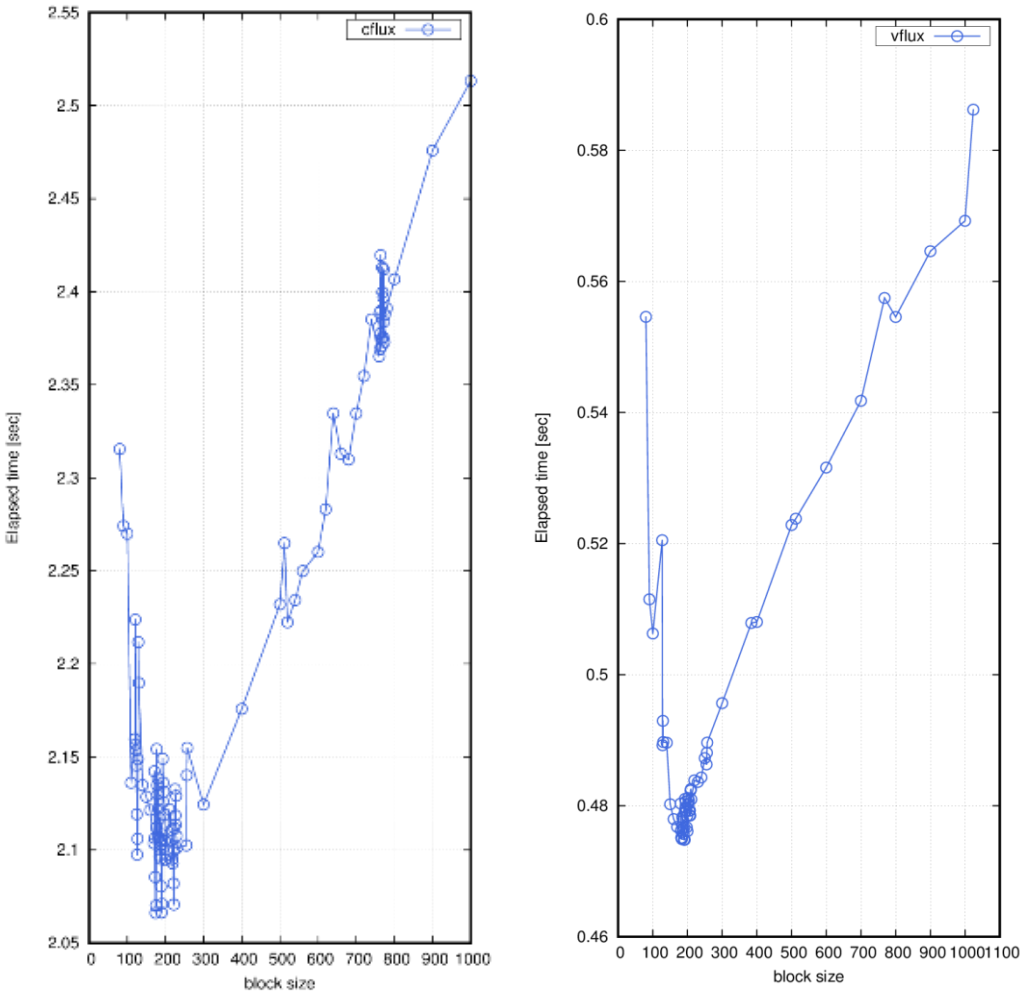
b）チューニング後

図 3 多重ループの実装

図 3b のプログラムで、多次元配列である a3d,b3d,c3d を、サブルーチン kernel で受けるときに、大きさ引継ぎ配列を用いることで、サブルーチン kernel 内では 1 次元配列として、長いループ長、ここでは N^3 の連続アクセスとしている（前述の(1)）。lb と l の 2 重ループ（ブロックの計算）および後半の 1 の 1 重ループ（ブロックの余り）がブロッキングを構成するループである（前述の(3)）。lbsize がブロックサイズとなり、もとのループ長（ここでは N^3）をブロックサイズで割った余りの処理が後半の l のループで実施されている。ブロックサイズはループを分割することで発生する分割ループ間で受け渡すデータが L2 キャッシュに収まるように調整する。計算部に対しては!ocl loop_fission_{target,

threshold}のコンパイラ指示行を入れることでループ分割を支持している（前述の(2)）。コンパイラ指示行で指定された「LS」および「50」は、分割アルゴリズムの種類と、閾値で、必要に応じて調整することが可能であるが、ここでは、デフォルト値で固定した。対象としたプログラムでは、対流項の計算は、MUSCL、SLAU、流束の差分計算を 1 つのループで実施しているが、このループがコンパイラの自動ループ分割では 57 個に分割された。粘性項に関しては、各物理量定義点で微分値を求めるループが 5 分割、粘性流束を求めるループが 25 分割、粘性流束の差分を求めるループが 7 分割に分割された。

図 4 にブロックサイズを決めるために行ったパラメトリック計算のうち、対流項および粘性項の計算結果を示す。これらの結果より、それぞれで最速となるブロックサイズとして、対流項では 174、粘性項では 192 を採用した。ブロックサイズを最適化することで、対流項で最大 25%程度、粘性項で最大 20%程度の高速化効果があった。

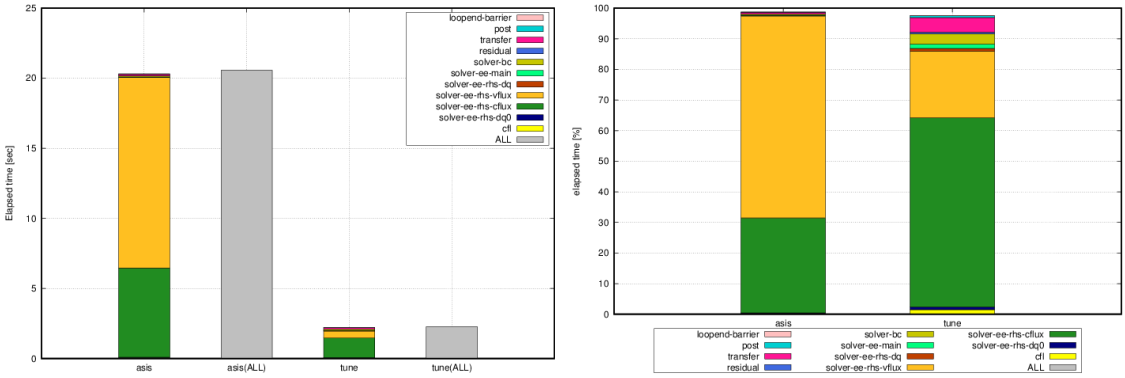


a）対流項　　b）粘性項

図 4 ブロックサイズの調査（横軸はブロックサイズ、縦軸は経過時間[秒]を示す）

4. FX1000 向け高速化チューニング結果

図 5 に高速化チューニング結果を示す。計測は 1 ノードで実施し、4 プロセス×12 スレッドを用いた。64^3 セルのブロックが 64 ブロック、セル数は約 1,700 万で、メモリ使用量は 10GB であった。ちなみに壁率は 0.18%である。図ではチューニング適用前後での全体および主要要素の経過時間や、主要要素の全体に占める割合を示している。



a）経過時間[秒]　　b）各要素の割合[%]

図 5 チューニング前後の比較

初期化を除いたプログラムのカーネル部分（図中で「ALL」で示す）では asis 版が 20.6 秒に対して、tune 版は 2.27 秒となり 9.1 倍の高速化となった。asis 版では一番時間がかかる粘性項の高速化が顕著で、tune 版では対流項よりも高速になった。この事は、経過時間全体に占める各要素の経過時間の割合を示す図 5 b)においても確認でき、現時点における tune 版で一番重い処理は対流項となっており、実行効率で 8%程度である。図 6 に tune 版対流項の CPU 性能解析レポートを示す。ループ分割とブロッキングにより L2 キャッシュがボトルネックになっていることが確認できる。L2 キャッシュの性能状況の評価を行った。tune 版で、最適化を抑止した状態でプロファイラー情報を取得し、浮動小数点演算数 112.75 [GFLOP]を求めた。また、メモリアクセス量は 852.52[GB]であった。この値と L2 キャッシュのミス数（3.5×10^8）×256[B]=89.6[GB]より、L2 キャッシュのアクセス量 852.52-89.6=762.92[GB]を求めた。これらよりプログラムのメモリに関する B/F：852.52/112.75=7.56、L2 キャッシュの B/F：762.92/112.75=6.77 をそれぞれ求めた。一方、H/W の B/F に関しては、メモリ：1,024/3379.2=0.30、L2 キャッシュ：3,600+/3379.2=1.06+、L1 キャッシュ：11,000+/3379.2=3.26 と求めた。これらプログラムの B/F と H/W の B/F の値とルーフラインモデルにより、理想的な実行効率を求めると、メモリ：0.30/7.56=3.97[%]、L2 キャッシュ：1.06/6.77=15.7[%]となる。メモリアクセスよりも性能は出ているが、L2 キャッシュアクセスを前提とした性能にはまだ到達していないことがわかる。過去のマシンなど経験的には対流項の実行効率は 15[%]程度と考えているので、更なるチューニングの余地があると考えられる。

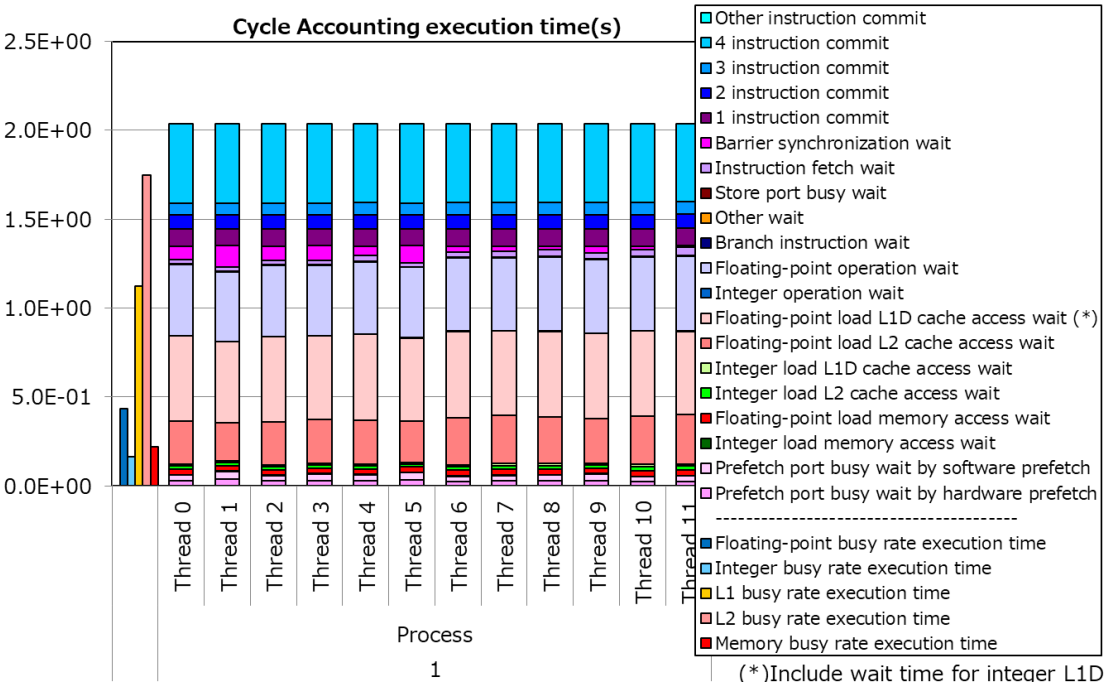
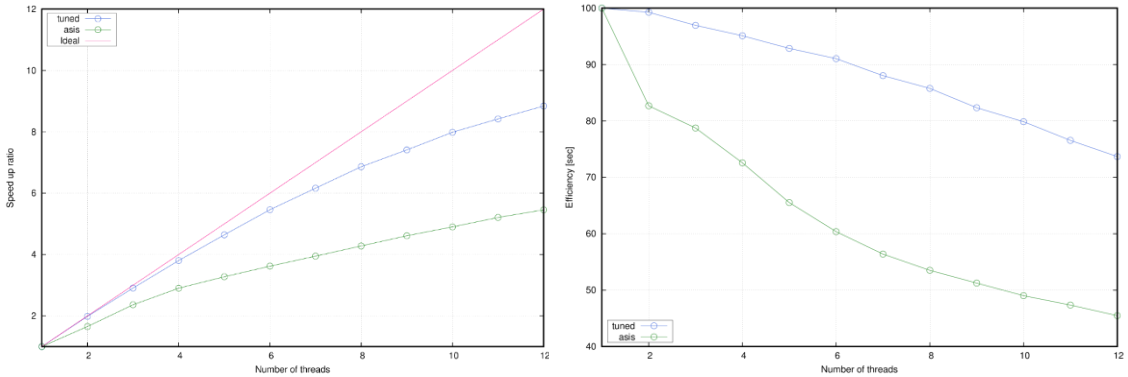


図 6 対流項のCPU性能解析レポート

図 7 にスレッド並列性能を示す。asis 版に比べて tune 版は全体的にスレッド性能が向上し、スレッド並列性能の効率は 12 コアで 72%となった。

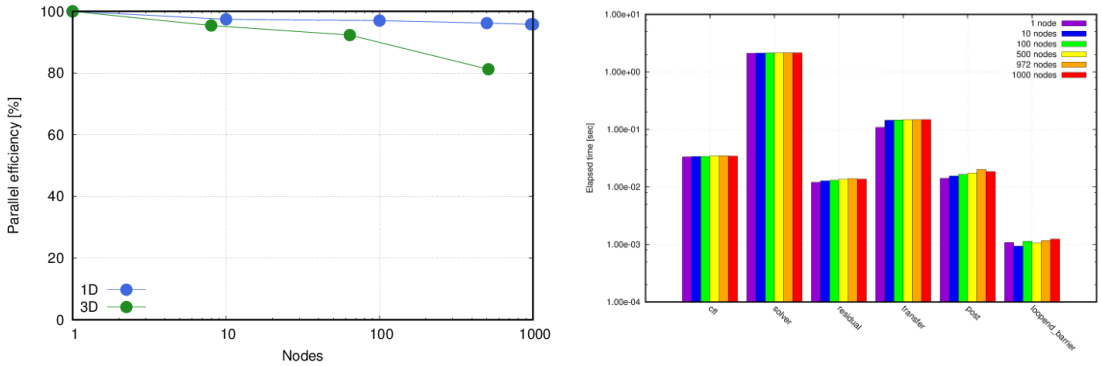


a）速度向上比（横軸はノード数、縦軸は速度向上比を示す）　b）並列効率（横軸はスレッド数、縦軸は並列化効率を示す）

図 7 スレッド並列性能

図 8 にノード間の並列性能を示す。ノード数の増加に合わせてブロックを拡張する weak scaling の結果で、「1D」は 1 次元方向に、「3D」は 3 次元方向にブロックを増やしている。多ノードにおいても良い並列性能を示していることがわかる。図 8b)に主要要素の経過時間を示す。プロセス間通信を含む処理（cfl, post, residual は allreduce、transfer

は irecv/isend/wait を用いた隣接通信、loopend_barrier は時間積分ループのバリア同期）はノード数の増加に伴い経過時間が延びているが、プロセス間通信を含まない処理である solver はほぼ一定であることが確認できる。なお、これらの時間計測においては、プログラムなどの初期化部分は除き、時間発展のループで時間計測を行った。その際に、多ノード、具体的には 500 ノードを越えた辺りから、ロードモジュールのファイル読み込み性能の影響が見られたため、時間発展の 1 ループ目は計測を行わず、2 ループ目から 11 ループ目までの 10 ループ分の計測を行った。



a）全体性能（横軸はノード数、縦軸は並列化効率[%]を示す）　b）主要要素毎の経過時間（横軸は各要素、縦軸は経過時間[秒]を示す）

図 8 プロセス並列性能

5. おわりに

FUJITSU Supercomputer PRIMEHPC FX1000 を対象に、CFD 構造格子プログラムの高速化チューニングを実施した。埋め込み境界法特有のループ内に存在する if 文への対応を行うことで、一般的な構造格子ソルバーに適用できる汎用的な手法について紹介しその効果を示した。ノード間の並列性能を weak-scaling で評価し、1,000 ノードまで良い結果が得られた。今後も引き続き更なる高速化チューニングを実施したいと考えている。

参考文献

1） FUJITSU Supercomputer PRIMEHPC, https://www.fujitsu.com/jp/products/computing/servers/supercomputer/

2） 高木亮治，杉崎由典，鈴木清文, “ステンシル系プログラムの低メモリバンド幅CPU向け高速化手法の検討”，第48回流体力学講演会/第34回航空宇宙数値シミュレーション技術シンポジウム論文集, pp101-106, JAXA-SP-16-007,2016.

3） 高木亮治, “CFDプログラムを用いたメニーコアCPUの特性評価”，第49回流体力学講演会/第35回航空宇宙数値シミュレーション技術シンポジウム, 2B13，東京, 6/30, 2017.

4） 高木亮治，河合宗司，久谷雄一，玉置義治, “階層型等間隔直交構造格子方の精度検証
—直交格子CFDワークショップ—”，第1回直交格子CFDワークショップ講演集,
pp77-88, JAXA-SP-20-006, 2021.

3.2 CityLBM コードの A64FX 向け最適化および NVIDIA GPU・AMD GPU に対する性能調査

日本原子力研究開発機構システム計算科学センター　小野寺直幸

1.　はじめに

汚染物質の大気拡散解析は、都市街区内の放射性物質の拡散予測に基づく核テロ対策や原子力発電所の廃炉作業で必要となる放射性物質拡散の事前評価など、原子力分野の課題解決に大きく貢献できる。この解析技術は、都市街区内の詳細な風況解析に基づくスマートシティ設計にも応用できるなど、幅広い工学分野への貢献も期待されている。都市部は様々な建物や構造物が林立した複雑な形状をしており、それらにより空気の流れ（風）が乱流状態になるため、時々刻々と変化する流れを予測できる高精度な数値流体力学（CFD）解析が必要である。また、それらの流れが人の生活圏に及ぼす影響を評価するためには、数 km の都市広域から数 m の細かな路地等をモデル化した大規模なマルチスケール解析が不可欠である。一方で、核テロ対策等の緊急時の汚染物質拡散予測には、実時間よりも速いシミュレーションの実現が求められている。

日本原子力研究開発機構（JAEA）の研究グループでは、それらの課題に対して、リアルタイムのマルチスケールシミュレーションの実現に向けて、都市の風の流れのスケールの違いに着目し、そのスケールに応じて計算格子の解像度を変化することができる適合格子細分化（AMR）法を用いた格子ボルツマン法に基づく CFD 解析コード CityLBM の開発を続けている。本報告書では、富岳等のスーパーコンピュータに搭載されている A64FX の活用に向けて、共同研究先である富士通株式会社と共に実施した最適化作業を記載する。また、A64FX と同様に、低消費電力・高性能アーキテクチャとして注目されている NVIDIA GPU・AMD GPU についても調査することで、次世代のエクサスケールスーパーコンピュータを想定した性能移植性について議論する。

2.　汚染物質拡散手法 CityLBM

CityLBM コードは、非圧縮流体の弱圧縮近似解法である LBM に基づいて風況解析を行う。LBM は連続体である流体に対して、格子上を並進・衝突する仮想的な粒子の集合と仮定し、格子上の粒子の速度分布関数について時間発展方程式を解く手法である。空間は等間隔の格子上で離散化され、有限個の速度を持つ粒子は並進運動により 1 タイムステップ後に格子点上に位置するため補間に伴う離散化誤差を含まない。CityLBM では、粒子の離散化速度モデルとして 3 次元 27 方向（D3Q27）モデルを採用すると共に、衝突過程は Cumulant モデル[M.Geier et al. Comput Math Appl 2015]を採用した。

従来の LBM では直交格子を用いた解析を行うが、CityLBM ではブロック型 AMR 格子に基づいた異なる解像度の格子が混在した解析を実現している。ブロック型 AMR 法では、領域毎の解像度の変化に対して、木構造の一つのブロックを同じ大きさの 8 つのブロ

ックに分割する八分木を採用することで表現すると共に、一つのブロック内に 64（4^3）個の格子を含むことで、格子上に定義された物理量に対して連続的なメモリアクセスを実現している。

CityLBM は C++言語を基に記述されており、CPU に対して OpenMP を用いたスレッド並列化、GPU に対して NVIDIA CUDA もしくは AMD HIP を用いたコア並列化を行うと共に、MPI を組み合わせることで分散計算機環境での高速化を実現している [N.Onodera et al. Boundary-Layer Meteorol 2021]。

3. CityLBM のデータおよびループ構造

LBM の主要な計算コストとして、粒子の速度分布関数 f の時間発展が挙げられる。D3Q27 モデルでは、27 個の速度分布関数が、ブロック内の 64（4^3）の格子上に定義され、それが木構造の数 N_{blocks} だけ確保される。速度分布関数の時間発展に関する疑似コード（C++言語のメモリ規則に従う）を以下に示す。

```
float *f[Nblocks][27][4][4][4], *fn[Nblocks][27][4][4][4]; // ダブルバッファリング用

for l in Nblocks {
  for k in 4 {
    for j in 4 {
      for i in 4 {
        fn[index] = operation( f, u, v, w, rho )
      }
    }
  }
}
```

ここで、変数 f・fn は、それぞれ読み込み・書き込み用の配列となり、格子点上の 27 個の速度分布関数に基づいた格子点毎に独立な計算 operation を行う。LBM の計算アルゴリズムの各アーキテクチャ向けの最適化として、再外のブロック（l）に対するループに、CPU のスレッド（OpenMP : #pragma omp parallel for）や GPU のブロック（CUDA : blockId.x）を、内側の 3 重ループに SIMD 命令（Intel OpenMP : #pragma ivdep）や GPU のスレッド（CUDA : threadId.x）を割り当てた。

4. A64FX 向けの最適化

A64FX 向けの C/C++コンパイラとして、A64FX の開発元である富士通がコンパイラも開発しており、それを利用することで高度な最適化命令である SVE（Scalable Vector Extension）が初めて利用可能となる。一方で、富士通コンパイラでは、正しくディレクティブ文を使用したとしても、GNU や Intel コンパイラで利用可能（もしくは最適化される）な機能が正しく動作しないため、それ特有の最適化が必要となる。以下に共同研究先である富士通と共に実施した A64FX 向けの最適化について示す。

i. SIMD 化対象のループ内に関数（もしくはインライン関数）が含まれる場合、パイプライン化や SIMD 化等の最適化が有効化されないため、手動にて関数をインライン展開（コピー＆ペースト）する必要がある。左記のインライン関数を使用した場合、コンパイラの出力にてインライン関数が展開されたことが出力されるが、最適化されていないため注意が必要である。
ii. 最適化対象の関数に条件分岐文が含まれる場合、パイプライン化や SIMD 化等の最適化が有効化されないため、手動にてビット演算に基づくマスク処理を実装する必要がある。
iii. ループ内の処理で複雑な処理を行うと、多くの場合でパイプライン化や SIMD 化等の最適化が有効化されないため、ループを複数に分割することが必要となる。
iv. 動的にメモリを確保する際に、aligned_alloc を利用することでメモリアクセスが高速化される。

性能移植性および開発効率化の観点から、元のプログラムを大きく改変する必要がある、最適化項目(i)「関数に対する手動インライン化が必須であること」、が致命的となる。左記のついて富士通に問い合わせたところ、「2022 年度 9 月現在では、富士通が開発するコンパイラの SVE 命令が有効とされる trad モードにおいて、C/C++のインライン関数の最適化を有効とする開発方針は無い」、と回答を得ており、最適化の項数と性能向上のトレードオフの観点が重要となる。

5. CityLBM コードの A64FX および GPU での性能調査

A64FX 向けに最適化した CityLBM の性能調査結果を報告する。計算環境として、名古屋大学のスーパーコンピュータ不老 Type I サブシステムを利用し、コンパイラオプションとして以下を使用した。

```
-O3 -std=c++11 -Kfast,parallel,openmp,optmsg=2 -Nlst=t -Krestp=all -Kocl
```

最初に CityLBM の主な計算時間を占めている streaming カーネルの A64FX 向けの最適化前および最適化後のメモリ帯域幅を表 1 に示す。性能測定は、A64FX の 1CPU に対して実施し、CPU 内に含まれるコアの集合単位である 4CMG に対して、それぞれ MPI プロセスを割り当てると共に、CMG に含まれる 12 コアに対しては OpenMP のスレッドを割り当てた。解析結果より、1CMG の理論メモリ帯域幅[GB/s]の 256 に対して、最適化前は 2.3、富士通との共同研究により行なった最適化後は 26.8 となり、10 倍以上の高速化が実現された。

表 1　streaming カーネルの A64FX の 1CMG 毎の計算性能

問題サイズ	4,096,000格子（64,000 × 4 × 4 × 4）
理論メモリ帯域幅[GB/s]	256
最適化前	2.3
最適化後	26.8

高速化率	×11.7

次に CityLBM コード全体の A64FX および NVIDIA GPU と AMD GPU に対する性能調査を行なった。A64FX に対しては最適版を、NVIDIA GPU および AMD GPU に対しては A64FX 向けの最適化をしていない同じコード（最適化前）を使用した。

表 2 の結果より、CityLBM の計算性能[MLUPS]（高いほど良い）は、A64FX で 10、NVIDIA V100 で 498、NVIDIA A100 で 732、AMD MI100 で 432 となり、A64FX で理論演算性能・理論メモリ帯域幅と比較して低い結果となった。一方で、NVIDIA や AMD の GPU では、理論性能から予測できる妥当な性能が得られた。また、プロセッサの性能評価として重要な指標である TDP 毎の計算性能[MLUPS/W]（高いほど良い）においても、A64FX に対して、NVIDIA V100 で 24.9 倍、NVIDIA A100 で 27.4 倍、AMD MI100 で 21.6 倍となった。

参考までに別の LBM モデルを採用している九州大学渡辺勢也氏の計算速度（A64FX 向けの最適化実施済み）を記載する[渡辺勢也、胡長洪、細分化格子法を導入したキュムラント型格子ボルツマン法の富岳での性能評価、計算工学講演会論文集 Vol. 27 (2022 年 6 月)]。TDP 毎の計算性能[MLUPS/W]では、A64FX に対して、NVIDIA P100 で 3.3 倍、NVIDIA V100 で 3.55 倍となった。

表 2　CityLBM の 1 プロセッサに換算した計算性能

1 プロセッサに換算した性能	Fujitsu A64FX（4 CMG）	NVIDIA V100（1 GPU）	NVIDIA A100（1 GPU）	AMD MI100（1 GPU）
理論演算性能（倍精度）[TFlops]	3.3	7.8	9.7	11.5
理論メモリ帯域幅 [GB/s]	1024	900	1555	1230
TDP [W]	150 *	300	400	300
CityLBM の計算性能 [MLUPS]	10	498	732	432
TDP 毎の計算性能 [MLUPS/W]	0.066	1.66	1.83	1.44
A64FX を基準とした TDP 毎の計算性能	1.0	24.9	27.4	21.6

*A64FX の TDP は公開されていなかったため同世代の Intel Ice Lake と同等と仮定

表 3　九大渡辺氏の LBM モデルの 1 プロセッサに換算した計算性能

1 プロセッサに換算した性能	Fujitsu A64FX（4 CMG）	NVIDIA P100（1 GPU）	NVIDIA V100（1 GPU）
理論演算性能（倍精度）[TFlops]	3.3	5.3	7.8
理論メモリ帯域幅 [GB/s]	1024	732	900

TDP [W]	150 *	250	300
LBM の計算性能 [MLUPS]	133.69	736.15	947.81
TDP 毎の計算性能 [MLUPS/W]	0.89	2.94	3.16
A64FX を基準とした TDP 毎の計算性能	1.0	3.3	3.55

*A64FX の TDP は公開されていなかったため同世代の Intel Ice Lake と同等と仮定

6. まとめ

本報告では、汚染物質拡散解析コード CityLBM の A64FX 向けの最適化、および、エクサスケールアーキテクチャの候補である A64FX、NVIDIA GPU、AMD GPU に対する性能を調査した。A64FX の最適化において、富士通の開発する C++コンパイラでは、GNU 等の一般的なコンパイラが実現しているインライン関数や条件分岐文の最適化が機能せず、手動による関数のインライン化やマスク処理が必要であることが確認された。以上の最適化を富士通との共同研究の下で実施した結果、オリジナルのコードと比較して、A64FX 上にて 10 倍以上の性能向上が得られた。一方で、A64FX と他のアクセラレータの性能比較を行なった結果、TDP 毎の計算性能[MLUPS/W]において、A64FX で 0.066、NVIDIA V100 で 1.66、NVIDIA A100 で 1.83、AMD MI100 で 1.44 となり、大きな差があることが確認された。また、別の LBM モデルにおいても、A64FX と NVIDIA GPU との TDP 毎の計算性能[MLUPS/W]が 3 倍以上となっており、GPU を用いた計算性能が高い結果となった。以上より、A64FX では、ブロック型適合細分化格子に基づく LBM においては十分な性能を得ることは期待できず、また、他の一般的なアーキテクチャ向けの開発環境と比較しても、C/C++向けのコンパイラが十分に整備されておらず性能移植性が低いと結論づけられた。

A64FX の理論性能と実際のアプリケーション性能の大きな乖離の原因については、共同研究先である富士通より報告する。

3.3 ジャイロ運動論的シミュレーションコード GKV の性能分析および OpenMP TASK 構文による通信・演算オーバーラップの検討

名古屋大学大学院理学研究科　前山伸也

1. はじめに

ジャイロ運動論的シミュレーションコード GKV は、磁場閉じ込めプラズマの微視的不安定性・乱流輸送現象を解析するために開発されたコードであり、「京」全系規模の良好な強スケーリングを利用して、マルチスケールプラズマ乱流物理の理解に貢献してきた。さらに将来の核燃焼プラズマを見通した新物理を開拓するためには、「富岳」の豊富な計算資源を有効に活用することが求められる。

GKV コードは、粒子の位置・速度の 5 次元位相空間を格子上に分割して、分布関数の時間発展を解く多次元格子法コードである。メモリ転送量と演算量の比は Byte/Flop～1 程度であり、単体性能としてはメモリバンド律速となる。また、高次元領域分割に起因する MPI ノード間通信も大きな割合を占めるため、良好な大規模スケーリングを達成すべく Tofu ネットワークにおける区分化プロセス配置や、OpenMP を利用した通信・演算オーバーラップが実装されている。

本報告では、富岳における GKV の性能分析およびチューニングを実施した結果についてまとめる。また、通信・演算オーバーラップの先進的な実装として、OpenMP TASK 構文によるタスク並列実装を検討した結果について報告する。

2. 「富岳」における GKV コードの単体性能分析とチューニング

「富岳」は「京」に比べて、ノード当たり 24 倍の演算性能を持ち、また、HBM のおかげでメモリバンド幅も 16 倍に向上しているため、CPU のメモリ Byte/FLOP 比は 0.38 の高い値を維持しており、メモリバンド幅律速のコードも効率的な動作が期待される。一方で、インターコネクトバンド幅は 2 倍程度の向上に留まっているため、インターコネクト Byte/FLOP 比は「京」の 10 分の 1 程度となっており、ノード間通信の比率を大幅に抑える必要がある。GKV は、「京」で既に演算とノード間通信が同程度で、通信・演算オーバーラップによる加速を実現していたが、「富岳」ではノード間通信の比率が過多になり性能劣化する懸念があった。そのため、新たに多粒子種衝突項陰解法モジュールを開発し、それにより演算比率の増大と求解時間の短縮を実現し、同時に実効演算性能の向上を達成した。こうした計算カーネルの変化も踏まえ、新たな演算ボトルネックの発見や「富岳」特有のチューニングの検討を行うため、GKV コード単体性能の詳細分析を実施した。

2.1 最適化指示行の追加

性能分析の結果、計算コストが高いサブルーチンは、新たに実装した多粒子種衝突項の差分演算を行う collision_full_ct であった。複雑な多数の差分項を計算するためにループボディが大きく、レジスタスピルが発生していることが判明した。「富

岳」ではレジスタ数が 32 に削減されているため、レジスタスピルが発生しやすくなり、ソフトウェアパイプライニング(SWP)が利きにくくなることは、「富岳」開発段階からすでに指摘されていた通りである。そこで、!OCL LOOP_FISSSION_TARGET(LS)の指示行による自動ループ分割を指定することで、レジスタスピルが減少し、SWP が促進された。ループ分割数について詳細な検討を進めた結果、作業配列のメモリアクセスに着目して手動で演算順序を並び替えてループ分割する方法が最も高速となった。加えて、ストラクチャーロードの影響を調査し、速度劣化を起こす部分には-Ksimd_nouse_multiple_structures 指定で抑制し、逆に高速化に寄与する部分には!OCL_SIMD_USE_MULTIPLE_STRUCTURES の指示行によるストラクチャーロードの有効化を施す最適化を実施した。

こうした大きなループボディにおけるレジスタスピルは計算コスト 2 位のサブルーチン collision_full_calc_moment でも見られたため、こちらもループ分割を実施した。また、サブルーチン collision_full_cf, collision_full_dt の他種粒子モーメント量との足し上げを行う部分でも同様のレジスタスピルが見られた。こちらに関しては、モーメント量が速度空間(iv)依存性を持たないことを利用して、iv ループ内で共通する因子を直前にまとめて計算して一時変数に格納することで、主たる iv ループ内でのレジスタスピルを解消することに成功した。

2.2 ループ内サブルーチンのインライン展開

ループ内のインデックス計算用サブルーチンコールのために SIMD 化できない箇所に対し、-x オプションを指定してコンパイラによるインライン展開を実施した。これに伴い、scatter store 命令により性能劣化していたが、ループ順序の入れ替えにより Multiple Structures 命令とすることで速度向上する箇所（vms2xy_pack_iz）も発見された。修正により、期待通りの性能向上が得られた。

2.3 OpenMP の環境変数設定

OMP_WAIT_POLICY=“ACTIVE”の環境変数を指定することで性能の改善が見られた。

2.4 配列式の Do ループへの書き下し

サブルーチン calc_error では、複素数配列の絶対値を計算する配列式があるが、その際 Gather load 命令が発行されている。これを手動で Do ループを用いて連続ロードとすることで性能向上が得られた。

2.5 チューニングによる性能改善効果

表 1 に実施したチューニングによる GKV コード単体性能の推移を示す。OCL 指示行の挿入やループ分割によりレジスタスピルが削減された効果（チューニング 1）により、経過時間が短縮されていることが確認できる。さらに、インライン展開による SIMD 化の促進（チューニング 2）や、OpenMP 環境変数の指定（チューニング 3)、配列式の do ループ化（チューニング 4）により、対象カーネル(colliimp)

はベースコードに比べて 1.23 倍の高速化が実現された。

表 1．チューニングに対する GKV コード単体性能の推移

分析観点	チューニング		timesteploop	colliimp
	チューニング対象	チューニング内容		
ベース	-	- RISTチューニング無しのコードをベースとする。 (gkvp_f0.61_nooverlap/)	16.81265	9.07795
(1)	コンパイルオプション + OCL指示行	✔ -Ksimd_nouse_multiple_structuresをベースに効果のあるループに!OCL SIMD_USE_MULTIPLE_STRUCTURESを指定	15.62537	8.06036
	collision_full_CT	✔ 主要ループを手動で2分割		
(2)	コンパイルオプション	✔ コンパイラのインライン展開によるSIMD化の促進 (-xGKV_colliimp.mxmy2ibuffiproc)	15.22112	7.55608
	vms2xy_pack_iz	✔ SIMD化により性能劣化するループにループ交換		
(3)	実行時環境変数	✔ 実行時にOMP_WAIT_POLICY="ACTIVE" の指定	15.04249	7.52774
(4)	calc_error	✔ 配列式のdoループ化	14.86216	7.38975

3. OpenMP TASK 構文による通信・演算オーバーラップの検討

単体性能分析により、SVE 演算や SIMD 命令といった基本項目が確認でき、FLOPS ピーク比としても 13.2%と悪くない値であることが確認できた。一方、通信・演算オーバーラップの同期制御の関係で単純なループ分割が適用できない箇所があるという問題も露わになった。通信・演算オーバーラップに関しては、現状一定の効果が確認できているものの、さらに柔軟な依存関係の管理、および通信同時発行による高速化の余地を議論するため、OpenMP TASK 構文による通信・演算オーバーラップについて検討を行った。プリ計算・MPI 通信・ポスト計算を複数回繰り返す、ごく単純なモデルプログラムに対して実装を行い、従来手法と比較を行った。

3.1 従来の OpenMP MASTER 構文による通信・演算オーバーラップの実装方法

まず、従来の MASTER 構文による実装手法について振り返る。演算と通信は順序に依存関係を持つが、例えば 5 次元空間のうちの独立な 1 つの軸について細分化することにより、複数の個々に依存する演算と通信のセットをループで処理する。この際、通信用作業配列などは 4 次元配列で済み、メモリ使用量削減にもつながる。さらに、このループを偶数と奇数に分けて作業配列をそれぞれ区別する。この上で、偶（奇）数の通信は MASTER スレッドで、奇（偶）数の演算はその他のスレッドで同時処理するようにすると、作業変数が途中上書きされることを避けつつ通信・演算オーバーラップが行える。この際、互いの通信・演算が終わるタイミングで!OMP BARRIER による同期をとって依存関係を担保する。良い点としては、同期制御についてはプログラムにより静的に決定されるため、依存関係管理のための計算コストなどは生じない。不便な点としては、意図しない同期を避けるために演算部分では非同期並列(!OMP DO と!OMP END DO NOWAIT)のみを用いる必要がある。また、マルチインターコネクトコントローラの特性として MPI 通信の同

時発行数が増えるとその恩恵を受けやすくなるが、BARRIER を挟んだ前後の MPI 通信が同時発行されるようなことは起こりえない。

3.2 新規検討した OpenMP TASK 構文による通信・演算オーバーラップの実装方法

OpenMP TASK 構文では、!OMP TASK に続く種々の構文により、各処理間の依存性を付随したタスクを発行する。依存性が解決され次第、実行可能なタスクを各 OpenMP スレッドが実行していく。今回の例でいえば、演算のタスクと通信のタスクがループ数分だけ発行され、偶数・奇数の同時実行可能な依存関係の記述を元に並列実行される。通信を発行するスレッドは MASTER スレッドに限らないため、異なる通信を同時発行するような状況も起こり得て、マルチインターコネクトコントローラの活用が期待される。

上記の演算と通信という単位でのタスク発行だけでは、1 スレッドが演算サブルーチンすべてを担当してしまい、メニーコアの恩恵を受けられないため、演算部分に!OMP TASKLOOP を用いることで、さらに子タスクに分割することが有効と考えられる。この方法は、子タスク間のみで同期がとれるので演算部のループ分割などにも柔軟に対応できるというメリットがある。

3.3 性能評価

テストコードに上述の OpenMP MASTER 構文または TASK 構文による通信・演算オーバーラップを実装し、経過時間を測定した。その結果、MASTER 構文方式では 0.587 秒に対し、TASK 構文では最も高速な場合で 0.738 秒と遅くなった。性能劣化の理由は TASK 構文のオーバーヘッドである。TASK 分割の粒度が細かい方がロードインバランスは均されるが、細かすぎるとオーバーヘッドが増大する。実際に、!OMP TASKLOOP において、タスクの生成数が多い場合は 0.958 秒、粒度を粗くすると 0.741 秒と高速化された。さらに極端な例としては、子タスクを生成しない場合の経過時間は 0.758 秒であり、子タスク生成のオーバーヘッドが回避されたためと考えられる。ただし、子タスクを生成しない場合は、12 OpenMP スレッドの内 3 つしか同時に動作していない（プレ演算、通信、ポスト演算）と考えられるので、それが子タスク生成よりも速いというのは、今回のサンプルプログラムでは設定された演算量が（タスク管理のオーバーヘッドに比べて）小さすぎたためと考えれられる。

4. まとめと今後の課題

第一に、「富岳」における GKV コードの単体性能評価を行った。その結果、「富岳」のレジスタ数 32 という制約に起因したレジスタスピルが発見された。OCL 指示行による自動ループ分割や手動でのループ分割により性能向上することが確認された。また、依存関係上まとめられる因子は事前に計算して一時変数に格納することで、演算数の削減とレジスタスピルの削減の両方に寄与することが確認された。コンパイルオプション指定のイン

ライン展開により SIMD 化が促進されることが確認された。全体のチューニングを通して 2 割程度の性能向上が実現された。結果的に、高い SIMD 発行率・SVE 演算率と 13%の FLOPS ピーク比が得られた。このことは、「京」に比べて高いメモリ Byte/FLOP 比は維持できたものの、インターコネクト Byte/FLOP 比は約 10 分の 1 と厳しい制約となる「富岳」において、演算密度の向上と求解時間の短縮を目指して GKV コードに実装された多粒子種衝突項陰解法が高演算効率を実現する演算カーネルとなっていることに由来する。以上の性能評価から、GKV コードの「富岳」での効率的な運用が確かめられた。

第二に、柔軟な依存性の管理と、通信の同時発行による高速化の余地を議論すべく、OpenMP TASK 構文による通信・演算オーバーラップの実装を検討した。今回用意したサンプルプログラムにおいては、TASK 構文のオーバーヘッドが大きく、従来の MASTER 構文による通信・演算オーバーラップよりも性能劣化するという結果になった。これは、小さな問題規模では TASK 構文のオーバーヘッドが顕著に現れうるということを示している。

更なる今後の追及の可能性としては、もしタスク構文のオーバーヘッドが大きい理由がキャッシュミスに起因するならば、事前にタスクサイズを調査してキャッシュミスしにくいように指定するなどのチューニングが可能かもしれない。また、TASK 構文のオーバーヘッドが問題にならない程度の演算負荷の大きな問題規模においては、BARRIER 同期を明示的に用いない TASK 構文による通信・演算オーバーラップの効率化や、通信の同時発行による通信高速化などの余地がありうると期待される。

4.1 ボクセル有限要素法におけるニューラルネットワーク型前処理カーネルの SIMD 化に関する検討

東京大学地震研究所　藤田航平

1. はじめに

近年のメニーコア CPU 計算機の性能を引き出すためには SIMD の有効活用が必須となっている。コンパイラの自動 SIMD 化など、一般に SIMD を活用する際には最内ループに SIMD をかけることが多いが、それ以外のループにおいて SIMD をかけることで高い性能が得られる場合がある。本検討においては、このようなカーネルの一例としてボクセル有限要素法における連立方程式ソルバー内で使われる前処理カーネルを取り上げ、最内ループ以外のループに SIMD をかけることで A64FX CPU 上で高速化される例を示す。ACLE intrinsics により最内ループ以外のループに SIMD を適用することで、コンパイラによる自動 SIMD 化で生成された最内ループに SIMD がかかるプログラムと比べて 5.8 倍の高速化が得られた。以下、対象カーネルの説明、性能計測の結果、および、本検討のまとめを示す。

2. 対象カーネル

有限要素法における線形方程式ソルバーにおいては行列ベクトル積カーネルが主要な計算コストとなるが、ランダムアクセス・データレカレンスが含まれるカーネルとなるため、近年の計算機で性能を出しにくい傾向にある。この線形方程式ソルバー内の計算を、近年の計算機に適したニューラルネットワーク型の計算に移すことで、計算を高速化する方法が考案されている。本検討では、そのような手法のうち、対象問題の物理的な特性を踏まえて構築した Green's function-based Neural Network (GF-based NN) preconditioner を活用する[1]の方法において主要な計算カーネルの一つとなる GF-based NN predictor kernel を対象とする。

GF-based NN predictor kernel のおいては、3 次元空間内において、対象節点の周りの情報に係数をかけて節点値を計算する構造となっており、対象節点のx,y,z方向に関する 3 重ループ内において、x,y,z方向の近隣節点に関する重みを足しこむ 3 重ループで計算する構造となっており、3 + 3 = 6 重のループとなっている(図 1)。計算はすべて単精度で実行される。周りの節点の情報(`rs`)をロードして係数(`cocs, we1~we8`)にかけ合わせ、節点値(`zs`)に足しこむこととなるが、この際、`i1,j1,k1` ループのループ長が 7 と短いために最内ループ(`i1`)に SIMD をかけると単精度 SIMD 幅(16 要素)の一部しか活用されず性能が出ないこととなる。そこで、ループ長が十分に長い `i` ループに SIMD をかけ、ロードする配列(`rs,cocs`)について最内に `i` に関する項を持ってくることでデータのロードを連続 SIMD 化し、また、`grs1,grs2,grs3` をレジスタ上に保持し続けつつ `i1,j1,k1` ループを実行することで高性能が得られると期待される。

```
                                    nef=3
                                    nd=3
                                    (コンパイル時既知)
1   do k=1+nef-1,nez+1-nef+1
2   do j=1+nef-1,ney+1-nef+1
3   do i=1+nef-1,nex+1-nef+1
4     we1=wei(1,i,j,k)
5     we2=wei(2,i,j,k)
6     …
7     we8=wei(8,i,j,k)
8
9     grs1=0.0
10    grs2=0.0
11    grs3=0.0
12    do k1=1,nd*2+1
13    do j1=1,nd*2+1
14    do i1=1,nd*2+1
15      rs1=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,1)
16      rs2=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,2)
17      rs3=rs(i1+i-nd-1,j1+j-nd-1,k1+k-nd-1,3)
18      cocs1=cocs(i1,j1,k1,1)
19      cocs2=cocs(i1,j1,k1,2)
20      cocs3=cocs(i1,j1,k1,3)
21      ww1=we1+we2*cocs1+we3*cocs2+we4*cocs3
22      ww2=we5+we6*cocs1+we7*cocs2+we8*cocs3
23      grs1=grs1+rs1*ww1*coe1s(i1,j1,k1,1)
24      grs1=grs1+rs2*ww2*coe1s(i1,j1,k1,2)
25      grs1=grs1+rs3*ww2*coe1s(i1,j1,k1,3)
26      grs2=grs2+rs1*ww2*coe2s(i1,j1,k1,1)
27      grs2=grs2+rs2*ww1*coe2s(i1,j1,k1,2)
28      grs2=grs2+rs3*ww2*coe2s(i1,j1,k1,3)
29      grs3=grs3+rs1*ww2*coe3s(i1,j1,k1,1)
30      grs3=grs3+rs2*ww2*coe3s(i1,j1,k1,2)
31      grs3=grs3+rs3*ww1*coe3s(i1,j1,k1,3)
32    enddo
33    enddo
34    enddo
35
36    zs(i,j,k,1)=grs1
37    zs(i,j,k,2)=grs2
38    zs(i,j,k,3)=grs3
39  enddo
40  enddo
41  enddo
```

図 1 . GF-based NN predictor kernel の構造。赤字で示された i ループに SIMD をかけることで、データアクセス・計算が連続となり高性能が期待される。

3. 性能計測

FX700 における富士通コンパイラにおいては最内ループにおいて自動的に SIMD がかかってしまい、図 1 のカーネルにおいて i ループに SIMD がかからない実装となってしまったため、ACLE intrinsic を用いて明示的に i ループを SIMD 化したカーネルを実装した。FX700 1 ノード(4 MPI process × 12 OpenMP threads)において得られた性能を表 1 に示す。As is は図 1 の Fortran プログラムを富士通コンパイラにおいてコンパイルして生成した実装、ACLE は ACLE にて実装したプログラムの結果である。表から、実行時間が 5.8 倍高速化され、高い性能(倍精度ピーク比で 49.7%, FP32 ピーク比で 24.8%に相当)が得られたことがわかる。このように、最内ループ以外のループに SIMD を適用することで高速化が得られる例があることが示された。なお、最内ループ以外のループに SIMD 化をかければどのようなループでも高速化できるわけではなく、該当のループに SIMD をかけた際

にデータアクセスが削減され、また、必要なデータアクセスが連続ロード・ストアとなるようなデータ構造となっていることが性能に直結するため、カーネル構造をそのような形に変形することが重要となると考えられる。

表 1　FX700 上での GF-based NN predictor kernel の性能計測結果

	Elapsed time (s)	TFLOPS	Peak to FP64
As is	0.3452	0.28	8.3%
ACLE	0.0598	1.68	49.7%

4.　まとめと今後の展望

本検討ではメニーコア CPU 計算機の性能を引き出すために必須となる SIMD の有効活用の一つの方法として、最内ループ以外のループに SIMD をかけることを検討した。ここでは、ボクセル有限要素法における連立方程式ソルバー内で使われる前処理カーネルを対象に、FX700 の A64FX CPU 上で動作するコードを開発し、最内ループに SIMD がかかる通常の実装との性能を比較した結果、5.8 倍の高速化が得られた。このように、SIMD を最内ループ以外のループに用いることで高速化が得られる例があることが示された。なお、最内ループ以外のループに SIMD 化をかければどのようなカーネルでも高速化できるわけではなく、該当のループに SIMD をかけた際にデータアクセスが削減され、またデータアクセスが連続ロード・ストアとなることが性能向上のために重要であり、このような形にアルゴリズム・カーネル構造を構築・変形することが重要となる。

本検討では、ACLE を用いて手動で SIMD を最内ループ以外のループにかけたが、将来的には、コンパイラにより自動的に最内ループ以外のループにも SIMD をかけることができるようになれば低い開発コストで性能向上が得られるようになると期待される。Intel Compiler による自動 SIMD 化においては最内ループ以外のループに SIMD をかける機能があり、富士通コンパイラにおいても類似の機能が実装されることで高い性能が得られる可能性がある。

[1] Tsuyoshi Ichimura, Kohei Fujita, Muneo Hori, Lalith Maddegedara, Naonori Ueda, Yuma Kikuchi, “A Fast Scalable Iterative Implicit Solver with Green’s function-based Neural Networks”, ScalA20: 11th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems@SC20

4.2 宇宙プラズマ2次元 Particle-In-Cell コード PIC2D の性能測定および推定

名古屋大学宇宙地球環境研究所　梅田 隆行

1. はじめに

希薄で無衝突状態にある宇宙プラズマは、磁気流体力学、イオン運動論、電子運動論などの様々な時空間スケールで物理現象を記述できる。Particle-In-Cell（PIC）法は、1960 年代にプラズマ粒子と電磁場との相互作用のシミュレーション手法として考案されて以来、様々な分野において幅広く用いられている。オリジナルの PIC 法は、プラズマ物理及び電波科学分野において開発された。格子点（Cell）上に定義された電磁場をマックスウェル方程式により解き進め、その電磁場 Cell 中を荷電粒子がニュートン・ローレンツの運動方程式に基づいて加減速を受けながら動きまわることから PIC と名付けられた（文末資料 p.1、以降ページ番号のみ表記する）。現実空間に存在する膨大な数の荷電粒子を有限の計算機資源で扱うことは不可能であるため、PIC 法では、ある程度まとまった数の荷電粒子の集団を 1 つの“超”粒子（super-particle）として扱っている。

PIC 法では、ラグランジュ変数である粒子の位置及び速度と、オイラー変数である電磁場がデータ配列として混在しているため、スカラ型超並列計算機において高い性能を得るのは容易ではない。そのため、様々なスカラ型 CPU における性能特性を理解しておく必要がある。位置及び速度は共に 3 次元であるが、既存のコンピュータシステムにおいて 6 次元問題を扱うのは容易ではないため、本プロジェクトで用いる PIC2D では実空間（位置）を 2 次元、速度空間を 3 次元とした 5 次元問題を扱う。本稿では、ループ分割を用いた性能チューニングおよび性能測定結果を示す。

2. プログラム概要

プラズマ粒子の位置$\mathbf{r}$及び速度$\mathbf{v}$は、以下の荷電粒子の運動（ニュートン・ローレンツ）方程式によって記述される。

$$\frac{\mathrm{d}\mathbf{r}_n}{\mathrm{d}t} = \mathbf{v}_n \tag{1}$$

$$\frac{\mathrm{d}}{\mathrm{d}t}(\gamma_n \mathbf{v}_n) = \frac{q_n}{m_n}(\mathbf{E} + \mathbf{v}_n \times \mathbf{B}) \tag{2}$$

また電場$\mathbf{E}$及び磁場$\mathbf{B}$の時空間発展は以下のマックスウェル方程式によって記述される。

$$\frac{\partial \mathbf{B}}{\partial t} + \nabla \times \mathbf{E} = 0 \tag{3}$$

$$\frac{1}{c^2}\frac{\partial \mathbf{E}}{\partial t} - \nabla \times \mathbf{B} = -\mu_0 \mathbf{J} \tag{4}$$

さらに、荷電粒子が作る電流密度$\mathbf{J}$を計算するために、以下の電荷の連続の式を用いる。

$$\frac{\partial \rho}{\partial t} + \nabla \cdot \mathbf{J} = 0 \tag{5}$$

運動方程式(1)及び(2)の時間積分が2次精度になるように、粒子の位置と速度は互いに$\Delta t/2$ずれた時刻に定義される。また粒子の位置と電場を同じ時刻に、粒子の速度と磁場を同じ時刻にすることで、Maxwell 方程式(3)及び(4)による電場の更新と磁場の更新がそれぞれ2次精度になるようにしている。電磁場の各成分$(E_x, E_y, E_z, B_x, B_y, B_z)$は、空間差分が 2 次精度になるよ

うに、Yee 格子と呼ばれる Staggered 格子上に定義される[1]。また、電流密度(J_x, J_y, J_z)は電場(E_x, E_y, E_z)と同じ空間格子上かつ粒子の速度(磁場)と同じ時刻に定義される。ラグランジュ変数である荷電粒子の位置及び速度(x, y, z, v_x, v_y, v_z)と格子状のオイラー変数である電磁場$(E_x, E_y, E_z, B_x, B_y, B_z)$が混在していることが、プラズマ PIC コードの特徴と言える。

プラズマ PIC コードのカーネルは3つに大きく分けることができる(p.2)。1つ目はマックスウェル方程式(3)及び(4)による電磁場の更新で、これには FDTD 法[1]が用いられる。電磁場更新カーネルの負荷が全体の計算負荷で占める割合は 1%未満であり、ここでは割愛する。

2 つ目のカーネルは、式(3)による荷電粒子の速度計算である(p.3 及び 4)。粒子の隣接格子に対する重みを計算し、荷電粒子の位置における電磁場の値を隣接格子から重みに基づいて計算し[2]、最後に Boris 法[3]に基づいて荷電粒子の位置における電磁場の値を用いて粒子を加速する。粒子の重みは 4 次多項式に基づく。

3 つ目のカーネルは、式(5)による電流密度の計算である(p.3 及び 5)。粒子の位置を更新するとともに隣接格子に対する重みを計算し、荷電粒子の位置における電流を電荷保存法[4]に基づいて隣接格子へ割り振る。粒子の重みは速度計算と同じ 4 次多項式に基づく。また、スレッド並列におけるリダクション演算を回避するために、マルチカラー法とループタイリングを併用している[5]。

速度計算カーネルと電流密度計算カーネルは、外側に格子に関するループ、最内側に粒子に関するループ構造を持つ(p.4 及び 5)。このようなループ構造を PIC コードで用いるには、粒子の配列を粒子の位置に基づいて並び替える必要があり[6]、スレッド並列計数ソート[7]を用いる。また、プロセス間のロードバランサーには Oh-Help[8]を採用している。

以下では、速度計算(velocity カーネル)と電流密度計算(current カーネル)の2つのカーネルについて性能測定を行った。

3. 測定環境

測定に使用した PIC コードは、新たに開発したものである。性能測定には、名古屋大学の「不老」スーパーコンピュータ FX1000 を使用した。なお、本測定は単一ノードで行い、2 プロセス・24 スレッドを利用した。またノード間の並列性能の測定は行わない。コンパイラオプションは以下のとおりである。

- `-Kfast,openmp,ocl -x250`

測定で使用した格子数は(nx,ny)=(1000,1000)、格子あたりの粒子数は nppc=135 であり、これは作業配列を含めて約 12GiB のメモリ量となる。

4. FX1000 におけるループ分割

まず、as-is コードをそのままコンパイルすると、以下のようメッセージがコンパイルリストの粒子のループで表示された(p.6)。

- `This loop cannot be software pipelined because of shortage of floating-point registers.`

粒子のループではこの他にも、レジスタスピルが発生していたため、まず、重み計算、粒子位置での電磁場計算、Boris 法による粒子加速の 3 つにループを分割した。これにより、重み計算計算のループはソフトウェアパイプライン化された。一方で、粒子位置での電磁場計算のループ

では、以下のようメッセージがコンパイルリストの粒子のループで表示された(p.7)。単純に電磁場 6 成分について、5x5 の格子点からデータを読んで重みを掛けるだけの演算であるが、ソフトウェアパイプラインのスケジューリングが不能で合った原因は不明である。このため、さらに細かくループ分割を行った。

- `This loop is not software pipelined because no schedule is obtained.`

Boris 法による粒子加速のループにおいてもレジスタ不足のメッセージが表示された(p.8)。これは、ローレンツ因子の計算における平方根や割り算の逆数近似がレジスタを多用するためである。このため、さらに細かくループ分割を行った。

最終的に、非常に細かいループ分割(p.9)を行うことにより、プログラム全体で計算時間を約7割に短縮することができた(p.10)。特に、計算時間が `velocity` カーネルにおいては約5割、`current` カーネルにおいては約6割となった。一方で、x86-64 プロセッサはループ分割を行わない as-is コードの方が高速であるという結果を得た。これは、ループ分割によって新たに発生した一時配列の更新が原因であると推測できる。

5. まとめ

ループ分割によって、プログラム全体において、約 1.4 倍の高速化がなされた。以下は、本 WG の活動で得られた知見である。

- レジスタ不足、レジスタスピル、スケジューリング不能などのループは分割することで、性能が改善する可能性がある。
- ただし、ループ分割はループ間のデータの受け渡しに一時配列を用いるため、x86 系の CPU では性能が低下する可能性もある。

参考文献

[1] K. S. Yee, Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media, IEEE Trans. Antennas Propag., Vol.AP-14, 302-307, 1966.

[2] I. V. Sokolov, Alternating-order interpolation in a charge-conserving scheme for particle-in-cell simulations, Comput. Phys. Commun. Vol.184, 320-328, 2013.

[3] J. P. Boris, Relativistic plasma simulation-optimization of a hybrid code, Proc. 4th Conference on Numerical Simulation of Plasmas, 3-67, 1970.

[4] T. Umeda, Y. Omura, T. Tominaga, H. Matsumoto, A new charge conservation method in electromagnetic particle simulations, Comput. Phys. Commun. Vol.156, 73-85, 2003.

[5] T. Umeda, Multicolor reordering for computing moments in particle-in-cell plasma simulations, submitted.

[6] T. Umeda, Paradigm shift in program structure of particle-in-cell simulations, Adv. Parallel Comput. Vol.36, 455-464, 2020.

[7] T. Umeda, S. Oya, Performance comparison of parallel sorting with OpenMP, Proc. 2015 Third International Symposium on Computing and Networking, 334-340, 2015.

[8] H. Nakashima, Y. Miyake, H. Usui, Y. Omura, OhHelp: A scalable domain-decomposing dynamic load balancing for particle-in-cell simulations, Proc. 23rd international conference on Supercomputing, 90-99, 2009.

4.3 宇宙プラズマ5次元ブラソフコード Vlasov5D の性能測定および推定

名古屋大学宇宙地球環境研究所　梅田 隆行

1. はじめに

希薄で無衝突状態にある宇宙プラズマは、磁気流体力学、イオン運動論、電子運動論などの様々な時空間スケールで物理現象を記述できる。Vlasov コードは、宇宙プラズマの第一原理である Vlasov(無衝突 Boltzmann)方程式と Maxwell 方程式により、プラズマ中の全ての電磁・静電波動と荷電粒子の運動との相互作用を解くシミュレーション手法である(文末資料 p.1、以降ページ番号のみ表記する)。Vlasov 方程式は、位置及び速度の関数として表される位相空間分布関数の保存則である。現実の世界では、位置及び速度は共に 3 次元であるが、既存のコンピュータシステムにおいて 6 次元問題を扱うのは困難であるため、本プロジェクトで用いる Vlasov5D では実空間(位置)を 2 次元、速度空間を 3 次元とした 5 次元問題を扱う。本稿では、A64FX プロセッサにおける OCL(Optimization Control Line)を用いた最適化および性能測定結果を示す。

2. プログラム概要

Vlasov コードは、Maxwell 方程式による電磁場の更新と Vlasov 方程式による分布関数の更新から成る。Maxwell 方程式は電磁場の解析で広く用いられている FDTD(Finite Difference Time Domain)法[1]により、その時間発展を解き進める。Maxwell 方程式の計算負荷は Vlasov 方程式の計算負荷に対して通常 0.1%未満であるため、今回は性能評価の対象外とした。

Vlasov 方程式による分布関数の更新は、演算子分離法により以下のように実空間(位置方向)の移流、電場による速度空間の移流及び磁場による速度空間の回転の3つの部分に分解され、それぞれの式が①実空間の移流(position カーネル)、速度空間の②移流(velocity_e カーネル)及び③回転(velocity_b カーネル)に対応する(p.2)。

$$\frac{\partial f_s}{\partial t} + \mathbf{v} \cdot \frac{\partial f_s}{\partial \mathbf{r}} = 0 \tag{1}$$

$$\frac{\partial f_s}{\partial t} + \frac{q_s}{m_s} \mathbf{E} \cdot \frac{\partial f_s}{\partial \mathbf{v}} = 0 \tag{2}$$

$$\frac{\partial f_s}{\partial t} + \frac{q_s}{m_s} (\mathbf{v} \times \mathbf{B}) \cdot \frac{\partial f_s}{\partial \mathbf{v}} = 0 \tag{3}$$

式(1)及び(2)は線形移流方程式であり、演算子"非"分離型の保存型解法[2]を用いる(p.5)。また式(3)は回転流方程式であり、back-substitution 法と呼ばれる演算子分離型の解法[3]を用いる。またこれらの方程式を解く上で、物理量の保存、解の無振動性、正値性の保証を満たす独自の保存型無振動スキーム[4,5]の 5 次精度版を用いている(p.3)。

速度空間の移流(velocity_e カーネル)の概要をプログラム1に示す(p.4)。実空間 x,y 方向の添え字を i,j、速度空間 vx,vy,vz の添え字を l,m,n とする。このプログラムでは、vx,vy,vz の各方向の1次元数値流束を計算したのち3次元数値流束を合成している(p.4)。分布関数データの定義は、モーメント計算(速度空間の積分)を高速に行うために f(vx,vy,vz,x,y)となっており、velocity_e カーネルでは数値流束データが l,n,m に依存するために 3 次元配列となる。一方、実空間の移流(position カーネル)では数値流束データが i,j に依存するために(l,m,n,i,j)の 5 次元配列となるべきであるが、分布関数デー

タを転置して f(x,y,vx,vy,vz)と定義することにより、2 次元配列となる。またこの配列の転置により、position カーネルは velocity_e カーネルとほとんど同じとなる(p.6)。

プログラム 1：velocity_e カーネルの概要

```
1  DO j=1,Ny
2    DO i=1,Nx
3      DO n=0,Nvz
4        DO m=0,Nvy
5          DO l=0,Nvx
6            dfx(l,m,n)=... !vx 方向のフラックスの計算
7            dfy(l,m,n)=... !vy 方向のフラックスの計算
8            dfz(l,m,n)=... !vz 方向のフラックスの計算
9          END DO
10       END DO
11     END DO
12     DO n=1,Nvz
13       DO m=1,Nvy
14         DO l=1,Nvx
15           f(l,m,n,i,j)=f(l,m,n,i,j)-dfx(l,m,n)+dfx(l-1,m,n) &
16                                    -dfy(l,m,n)+dfy(l,m-1,n) &
17                                    -dfz(l,m,n)+dfz(l,m,n-1)
18         END DO
19       END DO
20     END DO
21   END DO
22 END DO
```

3. 測定環境およびプログラムの特徴

測定に使用した Vlasov コードは、これまでの性能検討 WG 活動において FX100 に最適化されており、先に示したプログラム1を必要最低限の作業配列を用いて適切なループ分割を行ったものである。FX100 ではソフトウェアパイプラインを用いずにループアンローリングを行ったほうが高速であったため、ループの直前で以下の OCL を指定していた。

- !$OCL LOOP_NOFUSION NOSWP UNROLL(8)

性能測定には、名古屋大学の「不老」スーパーコンピュータ FX1000 を使用した。なお、本測定は単一ノードで行い、2 プロセス・24 スレッドを利用した。またノード間の並列性能の測定は行わない。コンパイラオプションは以下のとおりである。

- -Kfast,openmp,ocl -x250

測定で使用した格子数は(nx,ny,nvx,nvy,nvz)=(128,64,40,40,40)であり、これは作業配列を含めて約 12GiB のメモリ量となる。Vlasov コードで用いる次元数が多いため、1つの次元のループ長が短くなり、これによりコンパイラによる最適化の恩恵を十分に受けにくいという特徴を持つ。

OCL なし・ありの計測結果の比較より、FX1000 ではループアンローリングよりもソフトウェアパイプラインを用いたほうが高速であることが分かり、次節に示すように OCL の変更およびレジスタ使用率の最適化を行った。

4. FX1000 における OCL の変更および最適化

まず、ループの直前での OCL を以下のように指定した。

- !$OCL LOOP_NOFUSION SWP_POLICY(SMALL) SWP_FREG_RATE(n)

SWP_POLICY(SMALL)は短いループ長を想定したソフトウェアパイプラインを行う指示節である。また、SWP_FREG_RATE(n)はソフトウェアパイプラインに対してレジスタを n パーセント使用するかを指示する節であり、n は整数である。

富士通コンパイラは、最適化においてループ融合を多用する傾向にあり、またソフトウェアパイプラインが有効になるループの回転数がプログラムのループ長よりも大きくなる場合があるため(p.7)、上記の指示節が有効である。ただし、レジスタ使用率の調整は手動で行う必要がある。レジスタ使用率 n を調整する際には、以下の情報を参考にしたが、必ずしもこれらを満たすことが最適ではない。

- ITR(ループの回転数:number of iteration)がループ長よりも短い。
- IPC(サイクルあたりの命令数:instruction per cycle)が多い。
- SPILL(レジスタスピル)がない。

レジスタ使用率の調整により、プログラム全体で計算時間を 8 割に短縮することができた(p.8)。特に、velocity_b カーネルにおいては計算時間が約 7 割となった。一方で、position カーネルでは最内ループ長が 128 であり、レジスタ使用率の調整なしである程度の最適化が既に行われていたため、OCL あり・なしの差が小さかったと推測できる。

5. まとめ

OCL の最適化によって、プログラム全体において、約 1.2 倍の高速化がなされた。以下は、本 WG の活動で得られた知見である。

- FX1000 ではループアンローリングよりもソフトウェアパイプラインを用いたほうが高速である場合が多い。
- OCL によるレジスタ使用率の調整により、性能が改善する可能性がある。ただし、職人技である。

参考文献

[1] K. S. Yee, Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media, IEEE Trans. Antennas Propag., Vol.AP-14, 302-307, 1966.

[2] T. Umeda, K. Togano, and T. Ogino, Two-dimensional full-electromagnetic Vlasov code with conservative scheme and its application to magnetic reconnection, Comput. Phys. Commun., Vol.180, 365-374, 2009.

[3] H. Schmitz and R. Grauer, Comparison of time splitting and backsubstitution methods for integrating Vlasov's equation with magnetic fields, Comput. Phys. Commun., Vol.175, 86–92, 2006.
[4] T. Umeda, A conservative and non-oscillatory scheme for Vlasov code simulations, Earth Planets Space, Vol.60, 773–779, 2008.
[5] T. Umeda, Y. Nariyuki, and D. Kariya, A non-oscillatory and conservative semi-Lagrangian scheme with fourth-degree polynomial interpolation for solving the Vlasov equation, Comput. Phys. Commun., Vol.183, 1094–1100, 2012.

4.4 FMO プログラム ABINIT-MP の高速化と超大規模系への対応

立教大学理学部　望月祐志

1.　はじめに

フラグメント分子軌道(FMO)法は、北浦和夫氏によって 1999 年に創案[1]された、いわゆる分割＆統治系のアプローチの 1 つで、並列処理を駆使してタンパク質のような巨大分子系を量子論的かつ非経験的なフレームワークで扱うことが可能です[2-5]。タンパク質の場合、FMO 計算からはフラグメントとして分割された各アミノ酸残基の間、あるいは活性領域にはまりこんだ薬物分子と周囲のアミノ酸残基の間の相互作用エネルギーなどの数値指標が得られます。つまり、高速の量子化学的手法としてメリットだけでなく、計算対象の解析ツールとしても使えるのが FMO 法の大きな強みで、理論創薬や生物物理の分野で広く使われています[4,5]。

現在、FMO 計算のプログラムには書籍[5]内に記載されているように 4 つありますが、私たちが長年開発している ABINIT-MP[4,6]はスパコンとの相性が良い特徴があります。2020 年の 1 月に新型コロナウイルスのメインプロテアーゼに阻害剤 N3 が結合した結晶構造が公開（PDB-ID=6LU7）された後、退役間近の名古屋大学の FX100 を使って、すぐに FMO の計算を行って結果を解析し、3 月中旬にはプレプリントサーバ（ChemRxiv）で公開しました（後に正規論文として出版[7]）。このことがきっかけで、2020 年 4 月に理研 R-CCS からお申し越しをいただき、「新型コロナウイルス対策を目的としたスーパーコンピュータ「富岳」の優先的な試行的利用」の第 4 課題「新型コロナウイルス関連タンパク質に対するフラグメント分子軌道計算」の年間プロジェクトを行う機会を得ました。最初の一ヶ月で実行上のボトルネックを改善し、その後はプロダクションランを重ね、学会報告と論文出版[8-12]を行うことができました。改良した ABINIT-MP は、Open シリーズの Ver. 1 Rev. 22 として 2020 年 6 月 3 日にリリースしました。

ABINIT-MP の「富岳」での試行利用は、計算化学的には成果を多数もたらしたのですが、同時に問題点を強く認識する機会ともなりました。その問題点とは、高速化と超大規模系への対応の必要性です。そこで、プログラムの改良に際して計算機科学（HPC 分野）の専門家にコミットいただくことにしました。具体的には、名古屋大学の片桐孝洋研究室との共同研究を 2021 年度から始め、この SS 研の WG にも入れていただきました。こうしたことから、ABINIT-MP の Open シリーズも Ver. 1 系から Ver. 2 系に更新することを決めました。改良を進める対象プラットフォームとしては、大規模計算の容易さから「富岳」を第一に想定し、名古屋大学の「不老」 Type I や東京大学の「Wisteria/Odyssey」を含めた A64FX 向けとしました。公式には、JHPCN 課題の jh210036-NAH と jh220010 の枠で ABINIT-MP プログラムの改造作業を「不老」 Type I の上で進める形となっています（望月が課題代表者)。また、大規模なベンチマークテストや実証的な応用計算では「富岳」の HPCI 課題の hp210026 と hp220025 を使っています（こちらも共に望月が課

題代表者)。

以下、FMO 計算の概略を記した後、Ver. 2 系の最初のリリースである Rev. 4、2022 年 9 月時点でリリース準備を進めている Rev. 8 について、高速化と大規模系対応に話をフォーカスしてまとめます。

2. FMO 計算の概要

FMO 法では、対象分子系をフラグメントに分割します（以下、文献[1-5]を適宜参照のこと)。ここで、個々のフラグメントをモノマー、その 2 量体すなわち対をダイマーと呼ぶことにします。他のフラグメント系のアプローチと異なり、水素原子によるキャッピングを使わず、射影演算子を用いた技法（BDA; Bond-Detachment Atom）で結合の切断を行います。このため、タンパク質ではペプチド結合ではなく C_α 部位での分割になる点に注意が必要です（特に、ペプチド結合のカルボニル基が水素結合に関与する場合)。

FMO 計算の最初の段階は、与えられた基底関数の下、分子軌道法の基本近似であるハートリーフォック（HF; Hartree-Fock）によって各モノマーの分子軌道・電子密度を互いに静電ポテンシャル（ESP: ElectroStatic Potential）を自己無撞着的に課しつつ決定するモノマーSCC（Self-Consistent Charge）です。これによって、各モノマーに対して分極効果が導入されます。SCC 条件の達成までの大反復の回数は、系の荷電状態や構造に依存しますが、30〜70 回程度となります。次の段階では、モノマーで決せられた静電ポテンシャルを使ってダイマーでの HF 計算を行っていきます（SCC は課しません)。ダイマー計算によって、電子の非局在化効果が導入されます。高速化の手段としては、静電ポテンシャルの計算に Mulliken 電荷を用いる近似（ESP-AOC、ESP-PTC）が有効です。また、離れたモノマー間のダイマーは HF 計算をせずに静電的な寄与だけを評価（Dime-ES）することが常套で、ABINIT-MP では連続多重極展開（CMM: Continuous Multipole Moment）をさらに併用することも可能で、フラグメント数で千を超える大規模系の計算には特に有効です。

HF は平均場近似であり、電子間の静電的反発に起因する本質的な避け合いである電子相関は考慮されないため、通常は 2 次（MP2: second-order Møller–Plesset）の摂動論による電子相関エネルギー補正を加えて定量性を高めます。MP2 計算は、モノマーの場合には SCC 終了後各々に、ダイマーの場合は各 HF の後に続けて行い、加成性に基づいて HF エネルギーに補正されます。ABINIT-MP では、3 次の摂動計算（MP3）も MP2 の 2 倍程のコストで可能ですが[9]、メモリ要求はかなり増加します。MP3 計算の価値は、MP2.5 スケーリング（MP3 の相関エネルギーを 0.5 倍して加算）によって分子軌道法の中で最良の近似とされる 3 電子励起補正の 1,2 電子励起クラスター展開（CCSD(T)）並みの算定が可能となる点にありますが、MP2 でも部分再規格化（PR-MP2）を行うと素の MP2 よりも MP2.5 に近い値が得られます（コストの増加はありません）[9]。

3. 高速化と大規模系対応の第一弾：Ver. 2 Rev. 4

ABINIT-MP の並列化性能は既に十分に高い（並列化効率は 95.1%）ので、高速化に関しては演算部分の改良が直截となります。そこで先ず、計算レベルとしては最も利用頻度の高い FMO-MP2/6-31G*で、小規模で粒度の揃った Ala_9Gly をモデルとして、富士通社内の FX1000 の 2 ノードで 12 スレッド、8 プロセスの条件で、富士通の井上晃グループに Ver. 1 Rev. 22 のプロファイリングをお願いしました。その結果、2 電子積分の生成に全コストの 1/2 がかかり、HF 計算での添字処理と Fock 行列の構築が 1/4 コスト、その他が 1/4 コストとなることが分かりました。MP2 計算は、生成された基底関数添字の 2 電子積分の 4 回の線形変換が本質で DGEMM 処理が可能なことと、このテスト系のアミノ酸残基が小さいことから目立ちません。従って、2 電子積分の生成部部分が第一の加速の対象となりました。

ABINIT-MP の 2 電子積分生成ルーチン群は小原-雑賀の垂直漸化式関係（VRR）[13]に基づいた生成プログラムによって自動コーディングされており、{s, p, d, f}軌道の 4 つの組み合わせに応じて個別に生成されています。井上グループによるトライアルでは、s 軌道のみの(ss|ss)について一部スカラ変数化もしつつ SIMD 化を行い、加速が確認されました。また、OCL 指示詞の導入とコンパイラオプションの変更（Kfast など）のアドバイスもいただきました。これを受けて、ABINIT-MP の開発グループではコストが相対的に大きい(ss|ss)～(ss|sd)の 15 個の積分ルーチンに対して SIMD 化を手動で施しました（スカラ変数化も実施）。この結果、「不老」 Type I での測定で Ala_9Gly に対して 20%～30%の FMO-MP2 ジョブの時間短縮が得られました。

ここで、Ver. 2 系への更新のコンセプトについて記しておきます。Ver. 1 系では、専用の GUI（BioStation Viewer）を使い、単一構造での詳細な相互作用解析に力点を置くことを前提に整備されてきました。しかし、「富岳」の時代では古典力場（MM）による分子動力学（MD）シミュレーションによって生成された多数の液滴状のサンプル構造を一括して FMO 計算して統計的・データ科学的に解析することが常態化していきます[8,10]。何よりも問題となったのは、GUI 用にのみ必要なフラグメント数の自乗の大きさの配列を（計算の終了時にファイルにダンプするために）複数保持する必要があることでした。これにより、Ver. 1 Rev. 22 の段階では、「富岳」の上でも総フラグメント数の上限は 5.5 千程になっており、アミノ酸のフラグメント数だけで 3.3 千のスパイクタンパク質の計算[9]では水和モデルの計算が不可でした（水を入れると 1.5 万フラグメント程度になるため）。そのため、GUI への接続を放棄し、大規模系への対応を優先することにしました。また、統計的計算でのジョブ数は数百～千に達しますので、構造あたりの計算コストを下げることも重要となります。そのため、Ver. 2 系では 3 万フラグメントの扱い、A64FX 上の FMO-MP2 ジョブで Ver. 1 Rev. 22 に比して 3 倍の高速化を最終的な目標として設定しました。3 万フラグメントですと、例えば免疫グロブリンのタンパク質複合体が脂質膜部分も含めて扱えるため、生物物理学や分子生物学的にインパクトのある FMO 応用計算が可

能となります。

Ver. 2 系の最初のリリースは 2021 年 9 月 16 日の Rev. 4 で、上述の積分ルーチンの SIMD 化の他、MPI 通信量の低減、出力プリント量の抑制などの工夫により、基底関数と計算対象の系に拠りますが、FMO-MP2 ジョブの加速として Ver. 1 Rev. 22 に比して 1.2 倍～1.5 倍を得ました（「不老」 Type I と「富岳」で 2021 年夏にテスト）。6-31G*基底よりも、短縮長の長い cc-pVDZ の方が加速は顕在化する傾向も確認されました。大規模系の対応では、GUI 系配列群の削除の第一段階を済ませ、フラグメント総数 1.1 万のインフルエンザウイルスのヘマグルチニンの液滴モデル（タンパク質部分は 2.2 千フラグメント：PDB-ID=1KEN）の MP2 と MP3 ジョブの完走を cc-pVDZ 基底で達成しました。機能面の強化では、動的分極率のローカル版からの移植による「復活」、重要領域のみでの電子相関計算、機械学習用の（記述子）データダンプが追加されました。

2021 年度は、HPCI センターでのライブラリ整備（バイナリ提供）にも力を入れ、「富岳」、「不老」 Type I、「Wisteria/Odyssey」をはじめ、ほとんどの拠点に ABINIT-MP が導入されました。Ver. 2 Rev. 4 の高速化の改良は A64FX 向けですが、大規模系対応と機能強化は Intel Xeon 系でもメリットがあります。ただ、GUI を使った可視化解析にも依然として需要があることから Ver. 1 Rev. 22 も併存させる形としました。この併存は Ver. 2 系の更新が進んでも続ける予定です。

Ver. 2 Rev. 4 の改良と HPCI 拠点でのライブラリ化の状況は、文献[14,15]としてまとめて報告している他、学会でも発表しています。

4. 高速化の第二弾：仮 Ver. 2 Rev. 8

Ver. 2 Rev. 4 のリリースの後も、高速化の改良を続けており、2022 年度末にリリース予定の Rev. 8（仮）に反映させようとしています。2 電子積分周りでは、SIMD 化に加えて（レジスタスピルの低減のために）ループ分割を導入しています。積分のタイプにも拠りますが、積分生成ルーチンあたりで 20%～50%の加速が得られています。ループ分割の試行を含めた性能評価は、片桐研究室の学生の満田晴紀氏も実施されており、2022 年春の情報処理学会の HPC 部会の発表で学生奨励賞を獲得していることも書き添えます。

積分周りでは他に、井上グループからご指摘があった HF 計算での Fock 行列の構築から添字の等値性判断の if 分岐を排除する修正を行い、当該部分で 30%の加速を達成しました。その他、モノマーSCC の反復回数を低減することを意図してアンダーソン外挿を Fock 行列ベースから密度行列ベースに変更するオプションを設定しました。このオプションの挙動は、基底関数と対象系に依存して奏功する場合が多いですが、逆に回数が伸びて計算時間が長くなるケースもあります。その他には、プリントと通信量の追加の削減などを施しています。これらを反映させた 2022 年夏段階での加速では、FMO-MP2 ジョブで Ver. 1 Rev. 22 に比べて 1.5 倍～1.8 倍の加速となっています。

2022 年 9 月時点で試行を続けているのは、モノマーSCC 段階での 2 電子積分のバッフ

ァリングです。すなわち、生成された 2 電子積分をインコアで保持し、HF 計算/SCC 反復の間に保持しようという意図です。まだ暫定テスト段階ですが、これにより Ala_9Gly では 2 倍の加速となっています。実タンパク質では、Trp などの大きな残基が出てきますので、軌道タイプの組み合わせに応じて利用可能なメモリ量とのバランスを取って設定するのが現実的な使い方ですが、20 残基の Trp-Cage でも 2 倍の加速を得ており、本格的な利用の期待が持てます。2022 年 11 月には本実装の予定で、2022 年度末のリリースを目指す Ver. 2 Rev. 8 では Ver. 1 Rev. 22 比の加速で 2 倍を確実にしたい考えです。

より本格的な改造としては、Ver. 2 Rev. 8 には間に合わないですが、積分生成のルーチン群を小原-雑賀の VRR のみから Head-Gordon と Pople による水平漸化式関係（HRR）[16]を追加して併存させる変更を検討しています。HRR の方が VRR より総演算量を減らせるはずなのですが、ループの構造が変わるために最適化には VRR とは異なるアプローチが必要です。現在、HRR のプロトタイプを使っての評価を始めています。将来的には、VRR と HRR の各ルーチンを基底関数/軌道のタイプに応じて自動的に使い分けるような仕組みを導入することも考えています。

Ver. 2 Rev. 8 での大規模系対応では、液滴モデルのフラグメント化の扱いに工夫を凝らす方向で改善を図ります。タンパク質の水和では、表面の親水性のアミノ酸残基に近接する水分子は相互作用が大きく個別に扱うべきですが、離れた所に位置する水分子は 4 個～5 個でクラスターとしてまとめても影響は少ないはずです。この考えに基づき、PDB 構造ファイルを読み込んでクラスタリングを行う Python スクリプトを既に開発しています。この前処理により、3.3 千残基のスパイクタンパク質の液滴モデルの FMO-MP2 計算が実効フラグメント総数で 1.2 万程度で「富岳」の上で可能となる見込みです。

Ver. 2 Rev. 8 では機能追加も多くなります。注目領域でのイオン化エネルギーと励起エネルギーの計算機能のローカル版からの移植、分子凝集体での励起子ダイナミクスのための移動積分の算定、さらに相互作用解析の詳細化などが入りますので、リリース前のテストはかなり大変になりそうです。

5. まとめ

本報告では、2021 年度から実施している FMO プログラム ABNIT-MP の高速化と大規模化の改良についてまとめました。FMO-MP2 計算では 2 電子積分の生成と扱いがコストを決めていますので、積分生成ルーチンの SIMD 化とループ分割、HF での Fock 行列構築の改良などを手動で行いました。また、通信量やプリント量の低減なども実施しました。従前の Ver. 1 Rev. 22 に比して、2021 年 9 月 16 日にリリース済の Ver. 2 Rev.4 では 1.2 倍～1.5 倍の加速を得ました。また、Ver. 2 Rev. 8 としてリリースに向けて準備中のテスト版では 1.5 倍～1.8 倍の加速となっており、モノマーSCC 中の HF 計算での積分のバッファリングによって 2 倍に達する目処も立っています。大規模系への対応では、従前の 5.5 千フラグメントの 2 倍の 1.1 万フラグメントの扱いが可能となりました。文献[15]は、

Ver. 2 Rev. 4 の状況報告になります。

実証的なプロダクションランとしては、暫定テスト版を使って新型コロナウイルスやインフルエンザウイルスのタンパク質などに対して多サンプルの計算を「富岳」の上で進めており、改良の恩恵を自ら実感しています。

Ver. 2 系の A64FX 系での改良の最終目標は FMO-MP2 ジョブで従前比 3 倍の高速化と 3 万フラグメントの扱いとしています（実行環境は「富岳」を想定)。高速化では、積分生成ルーチンの VRR と HRR の併存化、SIMD 化やループ分割も必要です。また、大規模系対応では「非本質的な配列」の削除をさらに行うと共に、FMO 計算の実行に「本質的な配列」は、複数プロセス・複数ノードで分散する仕組みを導入することになりそうです。

6. 謝辞

ABINIT-MP プログラムの Ver. 2 系の研究開発は、中野達也氏（国立医薬品食品衛生研究所)、坂倉耕太氏（計算科学振興財団)、渡邊啓正氏（HPC システムズ）との共同作業として進められています。HPC の専門家の立場からコミットいただいている片桐孝洋先生にも深謝しますし、文献[15]として学際的な共著の報告ができたことをうれしく思います。また、タンパク質の液滴モデルの水のクラスタリングのスクリプトは、望月研究室の助教の土居英男氏によって開発されました。

「不老」 Type I の利用は JHPCN 課題の jh210036-NAH と jh220010、また「富岳」の利用は HPCI 課題 hp210026 と hp220025 に拠ります。

最後に、SS 研での活動では、富士通の事務局の皆様、井上晃氏のグループにお世話になりましたことに謝意を表して本稿を閉じます。

参考文献

[1] "Fragment molecular orbital method: an approximate computational method for large molecules", K. Kitaura, E. Ikeo, T. Asada, T. Nakano, and M. Uebayasi, *Chem. Phys. Lett.* **313** (1999) 701-706.

[2] *The Fragment Molecular Orbial Method: Practical Applications to Large Molecular Systems*, ed. D.G. Fedorov and K. Kitaura (CRC Press, 2009).

[3] "Exploring chemistry with the fragment molecular orbital method", D. G. Fedorov, T. Nagata, and K. Kitaura, *Phys. Chem. Chem. Phys.* **14** (2012) 7562-7577.

[4] "Electron-correlated fragment-molecular-orbital calculations for biomolecular and nano systems", S. Tanaka, Y. Mochizuki, Y. Komeiji, Y. Okiyama, and K. Fukuzawa, *Phys. Chem. Chem. Phys.* **16** (2014) 10310-10344.

[5] *Recent Advances of the Fragment Molecular Orbital Method - Enhanced Performance and Applicability*, ed. Y. Mochizuki, S. Tanaka, and K. Fukuzawa (Springer, 2021).

[6] "The ABINIT-MP Program", Y. Mochizuki, T. Nakano, K. Sakakura, Y. Okiyama, H.

Watanabe, K. Kato, Y. Akinaga, S. Sato, J. Yamamoto, K. Yamashita, T. Murase, T. Ishikawa, Y. Komeiji, Y. Kato, N. Watanabe, T. Tsukamoto, H. Mori, K. Okuwaki, S. Tanaka, A. Kato, C. Watanabe, K. Fukuzawa, in Ref. [5] pp. 53-67.

[7] "Fragment molecular orbital based interaction analyses on COVID-19 main protease - inhibitor N3 complex (PDB ID:6LU7)", R. Hatada, K. Okuwaki, Y. Mochizuki, K. Fukuzawa, Y. Komeiji, Y. Okiyama, and S. Tanaka, *J. Chem. Inform. Model.* **60** (2020) 3593-3602.

[8] "Statistical interaction analyses between SARS-CoV-2 main protease and inhibitor N3 by combining of molecular dynamics simulation and fragment molecular orbital calculation", R. Hatada, K. Okuwaki, K. Akisawa, Y. Mochizuki, Y. Handa, K. Fukuzawa, Y. Komeiji, Y. Okiyama, and S. Tanaka, *Appl. Phys. Express* **14** (2021) 027003-1-5.

[9] "Interaction analyses on SARS-CoV-2 spike protein based on fragment molecular orbital calculations", K. Akisawa, R. Hatada, K. Okuwaki, Y. Mochizuki, K. Fukuzawa, Y. Komeiji, and S. Tanaka, *RSC Adv.* **11** (2021) 3272-3279.

[10] "Dynamical Cooperativity of Ligand-Residue Interactions Evaluated with the Fragment Molecular Orbital Method", S. Tanaka, S. Tokutomi, R. Hatada, K. Okuwaki, K. Akisawa, K. Fukuzawa, Y. Komeiji, Y. Okiyama, and Y. Mochizuki, *J. Phys. Chem. B* **125** (2021) 6501-6512.

[11] "Fragment Molecular Orbital Based Interaction Analyses on Complexes Between SARS-CoV-2 RBD Variants and ACE2", K. Akisawa, R. Hatada, K. Okuwaki, S. Kitahara, Y. Tachino, Y. Mochizuki, Y, Komeiji, and S. Tanaka, *Jpn. J. Appl. Phys.* **60** (2021) 090901-1-5.

[12] "Collective residue interactions in trimer complexes of SARS-CoV-2 spike proteins on the basis of fragment molecular orbital method", K. Okuwaki, K. Akisawa, R. Hatada, Y. Mochizuki, K. Fukuzawa, Y. Komeiji, and S. Tanaka, *Appl. Phys. Express* **15** (2021) 017001-1-9.

[13] "Efficient recursive computation of molecular integrals over Cartesian Gaussian functions", S. Obara and A. Saika, *J. Chem. Phys.* **84** (1986) 3963-3974.

[14] "FMO プログラム ABINIT-MP の A64FX スーパーコンピュータ向け高速化と大規模化", 望月祐志, 中野達也, 坂倉耕太, *計算工学ナビ* Vol. **21** (2021 年秋号) 6.

[15] "FMO プログラム ABINIT-MP の整備状況 2021", 望月祐志, 中野達也, 佐藤伸哉 , 坂倉耕太, 渡邊啓正, 奥脇弘次, 大島聡史, 片桐孝洋, *J. Comp. Chem. Jpn.* **20** (2021) 132-136.

[16] "A method for two-electron Gaussian integral and integral derivative evaluation using recurrence relations", M. Head-Gordon and J. A. Pople, *J. Chem. Phys.* **89** (1988) 5777-5786.

4.5 乱流燃焼コード LS-FLOW HO 性能改善検討

宇宙航空研究開発機構　熊畑清

1. はじめに

一般的にアプリの性能チューニングというと，サブルーチン呼び出し等をもたない，コールツリーの低い階層にあるループが高コスト部分である際に，ルーフラインモデルに基づいて予測される演算性能 GFLOPS やメモリスループット GB/s に近づけるべく，例えば配列の次元を入れ替えてのキャッシュミスの低減や，スレッドのスケジューリング変更によるインバランス低減，リカレンスの除去などによる SIMD やソフトウェアパイプラインの促進などが一般的である．

一方で，我々のアプリでは，最内ループはごく小さなループであり，高コスト部位はそのような最内ループを含むような処理量としては小さなルーチンを多数・多階層に渡って呼んでいるような処理単位として大きな階層に位置するループである．性能改善にはそのような上位のループに対しての最適化がまず必要である．このような上位のループのボトルネック要因としては，キャッシュミスなどといったレベルよりもむしろ下位サブルーチン呼び出しのオーバーヘッドや，ソース上で処理全体が見通しずらいため同じ処理を重複して実行してしまうオーバーヘッド，そしてそもそも当該ループが最内ループでないことから，コンパイラによる最適化が適応されないということが挙げられる．本活動ではコンパイラによる最適化を望ましい位置で適応させるための改善について述べる．

2. 乱流燃焼コード LS-FLOW HO

液体ロケットエンジンの開発では，複雑な非定常燃焼場中で生じうる圧力変動が高周波かつ振幅の大きな振動に成長しエンジン部品の破損の原因となりうる燃焼振動の発生や，燃焼室壁面が高い熱流束にさらされることから起こり得る溶損などが高リスク要因として考えられる．これらの発生を設計時に評価するために Computational Fluid Dynamics(CFD, 数値流体力学力学)が重要な役割を担っており，JAXA では実機スケールの乱流燃焼シミュレーションを，高精度・高効率に行えるようにするために，Flux Reconstruction Method(FR 法，流束再構築法)[1]に基づく高精度な非構造格子乱流燃焼コード LS-FLOW-HO の開発を続けている[2]．

流体解析ではセルとセルとの境界上での Flux をいかに精度良く評価するかが重要であるが，FR 法では Fig.1 に示すように空間を分割した各セルに保存量および原始変数(ρ, u, v, w, P)などの値の定義点として Solution Point（SP）を定義し，セル内の物理量の分布を Lagrange 多項式で近似する．図中青い点が SP であり，赤い点はセル境界面に定義される Flux を定義する Flux Point(FP)である．SP 上での値によりセル内での値の分布を多項式補間する．補間次数 p に応じて SP と FP の数は変化する．

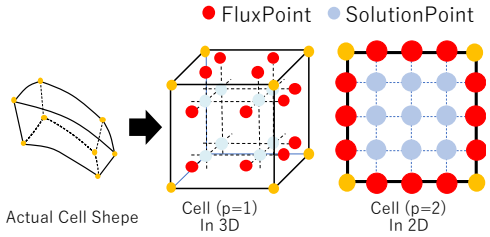


Fig.1 Cell shape and SP/FP distribution of Cell

セル境界での Flux は，セル境界を共有する二つのセル内の分布から近似 Riemann 解法により求める．Fig.2 に模式図を示す．図中実線が各セル内で Lagrange 補間された値の分布である．この分布はセル内では連続であるが，セル間で連続ではないのでので，図中赤点で示すように，セル境界を挟む 2 セルそれぞれから求めた値は一致しない．そこでセル境界上の流束を近似 Riemann 開放などの数値流束で一意に評価し，セル境界で連続になるように流束分布を修正する（点線）．

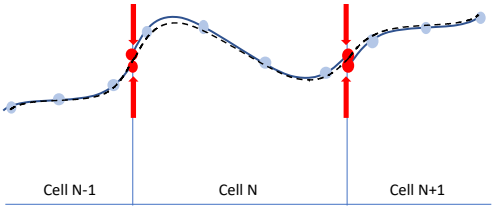


Fig.2 Calculating common flux of two cells

3. 性能上の課題

Fig.3 にプロファイラ fipp で採取した，ベンチマーク問題を実行した際の各処理のコスト分布を示す．ベンチマーク問題は総セル数 7100 からなる液体酸素と水素の同軸噴流火炎問題であり，水素と酸素の反応機構として 8 種の化学種を扱っており，補間次数は 2，つまりセル内の SP は xyz 各方向で 3 点づつの合計 27 点ある．

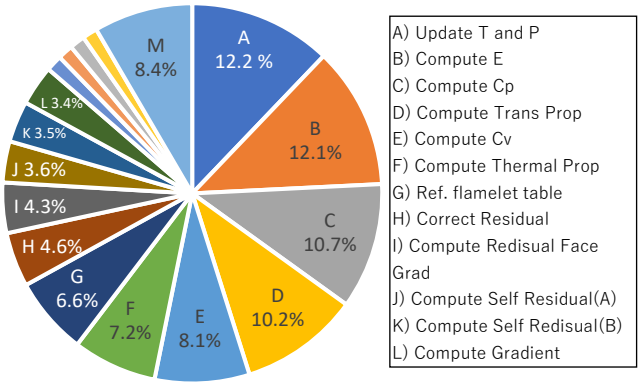


Fig.3 Cost distribution of benchmark problem

最も多くのコストを占めている区間 A)は温度と圧力の更新．ほぼ同程度のコストがかかっている B)はエネルギーの計算．以降熱容量の計算 C)と E)，輸送係数の計算 D)など，熱に関わる計算ルーチンが高コストになっており，熱物性値の計算 F）と，燃焼モデルとして一次元対向拡散火炎を解いた結果をテーブルとして参照する Flamelet Progress Variable（FPV）を用いているため必要な Flamelet Table の参照などで全体コストの 6 割強を占めている．

チューニングといえば通常は，下位のサブルーチン呼び出し等がないコールツリーの最も下位にあるループについて，GFLOPS や GB/s が出るようにするチューニングすることが一般的ではあるが，本コードでは高コストな処理は下位ループやサブルーチン呼び出しを含む比較的上位のルーチンであり，関数呼び出しのオーバーヘッドなど，演算待ちなどのボトルネック解消を行なうのが現時点での主眼となっている．

FX1000 上での性能上の大きな課題は，大きな回転数を持つ各処理のメインループから多くの下位サブルーチンを多階層にわたって呼んでおり，それら下位のループ中では化学種の数だけ回るループが多用されているため，最内ループの回転数が極めて小さいという点である．SIMD/SWP などのコンパイラによるループ最適化は最内ループに適用されるが最内ループが著しい低回転であるため効果がない．一例として上記ベンチマーク問題では 8 種の化学種を扱っているため最内ループの回転数は 8 である．次節ではこの問題を回避するための検討について述べる．

4. より上位の回転数が多いループでの SIMD 化

Fig.4 左に前述 Fig.3 における最高コスト区間 A) Update T and P のループ構造を示す．最上位のループはセルを巡るループでスレッド並列化されている．問題依存ではあるがここは十分な回転数がある．その内側のループがセル内の SP を巡るループとなっており，セル内の SP ごとに独立した計算を実行しており，種々の下位サブルーチンを呼んでいる．この中のニュートン反復のサブルーチンが最も高コスト処理であることは分かっており，詳細な情報を得るため FAPP 採取を試みたが，FAPP_START と FAPP_STOP で挟んだ部分の 1 回あたりの実行時間が 150us 以下であるため FAPP では正確に採取できなかった旨警告が出るためソースを Fig.4 右のように，セルを巡るループを 3 つの処理ブロックでループ分割して FAPP 採取を行った．なおループ分割しても性能に変化がなかったことは確認している．

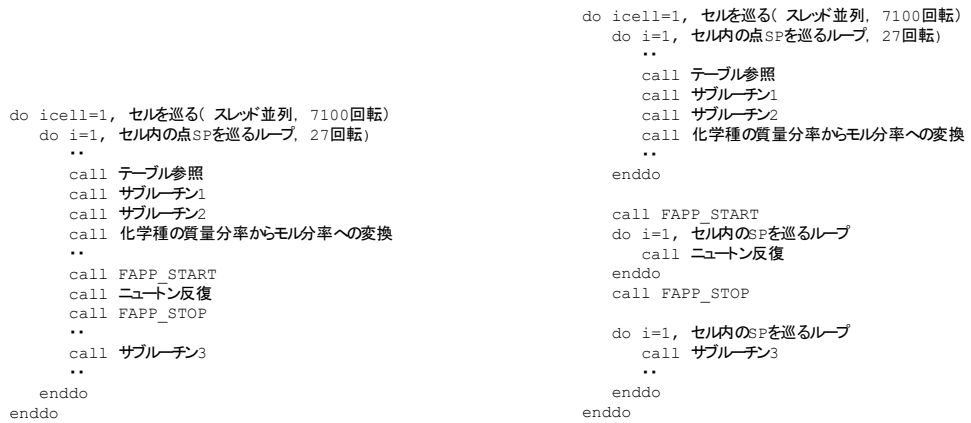


Fig.4 Typical loop structure
Left) Original, Right) Rev.1

Fig.5 左に FAPP の結果を示す．浮動小数点演算待ちの割合が高い．ニュートン反復ルーチン内部は，4 回転固定の反復ループを持ち，さらにその中に最内ループとして 8 回転のループを持つ構造になっている．コンパイラによる最適化として最内の 8 回転ループには SIMD が適応されているが，ソフトウェアパイプラインは適応されず，それら 4 回転ループ並びに 8 回転ループの外での演算も多いことからルーチン全体での SIMD 率は 40%程度であった．ここではまず，SIMD をより上位のループである「セル内の SP を巡るループ」に適応させるため以下の 2 点の変更を施した．

- 当該ループを最内ループとするため，ループ内でのサブルーチン呼び出しを全てループ中に直接実装(インライン展開だが，コンパイラの機能によるものではなく，手動にて実施)
- 当該ループに含まれるようになった回転数が少ないループを全てインライン展開(こちらは OCL UNROLL('full')の指定による)

これにより，目的のループで SIMD は適応された．しかし当該のループは，中に含む小回転ループが全てアンロールされ長大なループとなったため必要レジスタが多くなり，レジスタ不足によりソフトウェアパイプラインは適応されなかった．またレジスタのデータのキャッシュへの退避・復帰が発生し，その間演算器が待ち状態となり性能劣化してしまう．

このような場合，長大なループを適切な場所でループ分割し，ループ中の必要レジスタ数を削減しソフトウェアパイプラインを適応させることが有効であり，次の 3 つのループ分割手法，①コンパイラによる自動ループ分割，②コンパイラによる半自動ループ分割，③完全手動ループ分割を試みた．①は分割対象のループに OCL LOOP_FISSION_TAGET

(及び関連パラメータを指定する OCL)を指定する方法であるが分割されなかった．②は分割対象のループ中の分割したい位置に OCL FISSION_POINT を指定する方法であるが分割されなかった．①②でなぜ分割されなかったのかはコンパイラからのメッセージが不足しており不明である．③の完全手動ループ分割では，ニュートン反復ルーチン内の 4 回転する反復ループをアンロールしていることから，1 反復分の処理を区切りとして 4 分割することとした．

分割されたループの必要レジスタ数は削減されたはずであるが分割の度合いが不足していたのかレジスタ不足によりソフトウェアパイプラインは適応されなかった．完全手動のループ分割では，ループ間での計算結果を受け渡す一時配列の扱いが必要で煩雑であるため，より分割の度合いを細かくするのは現状はペンディングしている．

ソフトウェアパイプラインは適応されなかったが，SIMD がより上位のループで適応されるようになり，また分割によりループ当たりのレジスタ使用量が減り spill/fill が削減できるようになったことから，Fig.5 右に示すように実行時間はおよそ半減することができた．この時の SIMD 率はおよそ 40%から 90%へと向上している．ここで述べたループ構造は本アプリの他のルーチンでも頻出しているため，ソフトウェパイプラインが適応されないことに目をつぶれば，「セル内の SP を巡るループ」内のサブルーチン呼び出しは全てループ中に直接記述し，さらに小回転のループのアンロール，ループ分割によるレジスタ退避・復帰の抑制は効果がある．

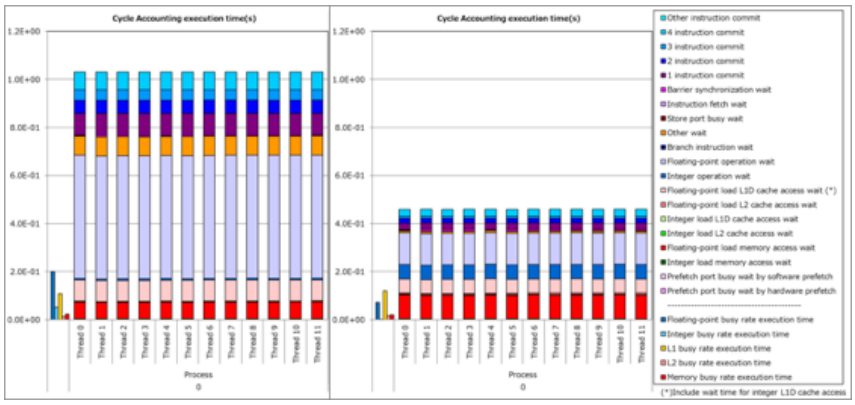


Fig.5 FAPP result
Left) Rev.1(=Original), Right) Latest

5. 結言

本稿では，JAXA で開発している乱流燃焼ソルバーLS-FLOW HO について，FX1000 上での性能チューニングの活動と成果の一部について述べた．SIMD を有効活用するにはなるべく長いループが SIMD 化対象となる必要があり，インライン展開とアンロールを適応した．一方でソフトウェアパイプラインも重要であるが，対象としたループはボディが大き

く必要レジスタ数が膨大であったため，ループ分割を施して必要レジスタ数を削減したがソフトウェアパイプラインの適応に十分なほど削減ができず，ソフトウェアパイプラインの適応はできなかった．ここで述べた以外のチューニングについては参考文献(3)で述べている．

また，上記で述べた以外にも，Fig.3 の G) Ref. flamelet table の処理では最内ループに if が含まれており，これも実行性能が低い要因となっており，if の mask 化などの効果を確認したい．今後コンパイラの改善に期待したいこととして，コンパイラからのメッセージが不足しており，狙っている最適化が適応されない際の原因がわからないので，より詳細なメッセージを出力できるようになることを期待したい．また前節では，ループにソフトウェアパイプラインを適応する試行錯誤を行なったが，ソフトウェアパイプラインの適応はできなかった．ソース全体を見渡してもソフトウェアパイプラインが適応されているループは，ほんの数行の演算のみを含んでいるような，ごく単純なループだけである．京や FX100 と比べて低下したレイテンシの影響を避けるにはソフトウェアパイプラインが必須であるため，ソフトウェアパイプライン化支援の拡充を期待したい．

参考文献

1) H. T. Huynh, “A Flux Reconstruction Approach to High-Order Schemes Including Discontinuous Galerkin Methods,”, 18th AIAA Computational Fluid Dynamics Conference Miami, Florida.
2) 芳賀臣紀，福島裕馬，熊畑清，根岸秀世，清水太郎, “液体ロケットエンジンの実機スケール燃焼器 LES に向けた取り組み”，日本燃焼学会誌, 64 巻, 208 号, pp.126-135, 2022.
3) 熊畑清，芳賀臣紀，高木亮治, “流束再構築法に基づく高次精度・非構造格子ソルバの FX1000での高速化事例”，第35回数値流体力学シンポジウム, E07-4, 2021.