第57回流体力学講演会/
第43回航空宇宙数値シミュレーション技術シンポジウム
2025年7月2日～4日
タワーホール船堀（東京）

1B01　ポスト富岳時代に向けた圧縮性燃焼ソルバLS-FLOW-HOのGPU化

○渡部修，芳賀臣紀，高木亮治（宇宙航空研究開発機構）

Towards GPU Implementation of a high-order combustion solver LS-FLOW-HO in the Post-Fugaku Era

Osamu Watanabe, Takanori Haga and Ryoji Takaki (JAXA)

Key Words : HPC, Performance evaluation, CFD, Flux reconstruction, High-order, GPU, OpenACC

Abstract

Future high-performance computing (HPC) systems, such as the post-Fugaku and JAXA's next-generation supercomputers, are expected to adopt GPU-based architectures. Adapting HPC applications to these accelerators is becoming increasingly important. We are porting a high-order combustion solver LS-FLOW-HO to GPUs using OpenACC. This report outlines the current progress, highlights key challenges in GPU adaptation - such as memory handling and parallelization - and shares insights gained through this effort.

１．はじめに

ポスト富岳 (富岳NEXT) やJAXAの次期スーパーコンピュータシステムをはじめとする次世代 High Performance Computing (HPC) システムでは，計算ノードにアクセラレータ，特にGPUを搭載した計算アーキテクチャが主流になることが予想される．2024年のSC24で発表されたTOP500では上位10システムの内，9システムがGPUを搭載したシステムであった[1]. こうした状況において, 既存のHPCアプリケーションをこれらのアクセラレータを効率的に活用できるようにすることが，喫緊かつ不可欠な課題となっている．

この課題に対し我々は現行のJAXAスーパーコンピュータシステムJSS3（JAXA Supercomputer System generation）[2]の利用段階から圧縮性燃焼ソルバLS-FLOW-HO[3]のGPU化を行ってきた[4]．GPU化はNVIDIAのGPUを対象にOpenACCを用いて行った．OpenACCはディレクティブベースでのアプローチであることから既存のコードへの適用も容易であり，GPU化を効率的に進めることができる．今回はこれまでLS-FLOW-HOに適用してきたOpenACCによるGPU化施策を現行のLS-FLOW-HOに適用する取り組みを行った. 現行のLS-FLOW-HOは, GPU化を適用した当時のバージョンから様々な機能追加および改修がなされているがGPU化施策は未反映であったため，改めて現行バージョンへのGPU化を行った．

先に述べたように，我々を取り巻く計算環境がGPU主流になることは容易に予想され，GPU化の経験が少ない開発者・研究者も自分たちのプログラムのGPU化に取り組みことが避けられない状況になりつつある．本稿では，GPU化の経験の少ない状態で本取り組みを実施し，GPU化を進める中で得られた知見や経験を共有する．

２．高次精度燃焼ソルバLS-FLOW-HO

支配方程式は圧縮性Navier-Stokes方程式である．燃焼モデルにはLaminar Flameletアプローチを採用し，事前に作成する化学種組成のテーブルは，混合分率と反応進行度をパラメータとするFlamelet Progress Variableモデル[5]を用いる．したがって，LESソルバで解く方程式は，密度，運動量，全エネルギーの保存則に加え，燃料と酸化剤の混合分率および反応進行変数の輸送方程式である．液体ロケットエンジン燃焼器の解析で重要となる高圧・極低温状態における実在流体効果を考慮するため，Soave-Redlich-Kwong（SRK）状態方程式[6]を用いる. 極低温状態の粘性係数・熱伝導係数については，Chungらのモデル[7]を利用する．

基盤となる離散化スキームとして高次精度のFlux Reconstruction（FR） 法[8]を採用する. FR法では計算セル内に高次精度化のための自由度を持つことにより，格子品質に依らずに任意の空間高次精度化が可能という点が最大の特徴である．FR法は非構造格子にも拡張可能であるが，本研究では実装が容易な六面体のセルを用いる．各セル内に解の定義点（Solution Point, SP）を導入して多項式近似を行う．デカルト座標の各軸方向のSPをK個とすると，K−1次の選点多項式によりセル内近似を得る. K−1次の多項式近似により空間K次精度のスキームとなる. FR法ではスキームの次数が大きいほどセル内のSP数が多く

なり演算数が増加するとともにセル当たりのデータ量も増加する．SPの選び方には任意性があるが，非線形対流項の誤差が小さいガウス求積点を用いる．

図1にこのコードの代表的なループ構造を示す．最外ループがセル数のループであり，その中に各セル内のSPの数のループ，最内ループとして方程式の成分数のループで構成された多重ループ構造となっている．このループ構造では最外ループのループ長が最も長く，セル数がループ長となっている．この最外ループはループ長が最も長いことに加えて依存関係もないため並列化に最も適したループと言える．そのため，LS-FLOW-HOのGPU化は基本的にセル数のループに対して行われている．

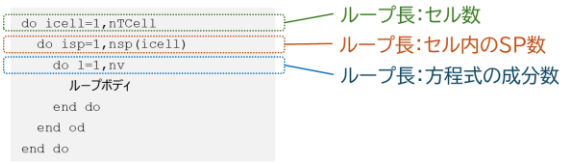

図1　LS-FLOW-HOの代表的なループ構造．

３．燃焼版LS-FLOW-HOのGPU実装

本節では既にGPU化されているLS-FLOW-HO（以降，GPU版LS-FLOW-HOと記す）において施されたGPU実装内容について説明する．GPUを用いた計算ではCPUホストのメモリとGPUデバイスのメモリ間のデータ転送が性能ボトルネックとなりやすい．芳賀・多胡によるGPU化においてはこのデータ転送が最小限となるよう，計算に必要な全ての配列変数をメインループに入る前にデバイスメモリに転送するようにした（図2）[4]. なお，こうした実装の場合，GPUメモリの容量により実行可能な計算格子規模が決まるため，メモリダイエットの検討を課題として挙げているが，CUDA Aware MPIを用いて複数GPUによる並列実行を可能とすることにより，更に大規模な計算の実行を実現している[4].

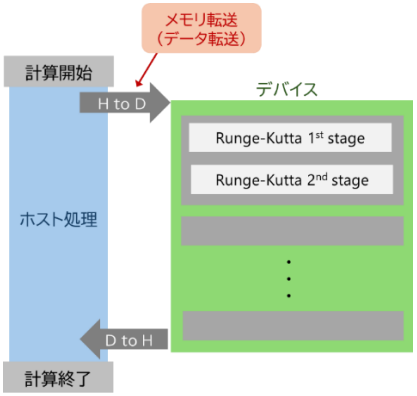

図2　GPU版LS-FLOW-HOのメモリ転送（参考文献[4]の図2を元に作成）．

本稿では，メインループ中にあるルーチンの内，GPU化のために使用した評価ケースにおいて呼び出されるルーチンをGPU化の対象とした．言い換えると，この評価ケースで通過するメインループ内の計算は実質すべてGPU化を施した．図3はRunge-Kutta 1 ステージ内のFRアルゴリズムを示す．なお，熱力学変数および輸送係数の計算はRunge-Kuttaの各ステージの外側で行われている．本稿の性能評価においてはLS-FLOW-HOのメインループを対象とした．また，現行版のLS-FLOW-HO（以降，現行版LS-FLOW-HOと記す）のGPU化を行うにあたり，基本的に芳賀・多胡が実施したOpenACC施策をそのまま適用した．これは今回GPU化を進める際に実行確認に用いた問題が芳賀・多胡がGPU化を行う際に用いた問題と類似しており，通過するルーチンおよびルーチン内の処理がほぼ同じであるためである．

$\boldsymbol{U}^f$ =Reconstruct($\boldsymbol{U}$)

$\boldsymbol{Q}$=Gradient($\boldsymbol{U}$)
$\boldsymbol{F}^f, \boldsymbol{R}$= Self_Flux($\boldsymbol{U}$)
$\boldsymbol{U}^f$=BC($\boldsymbol{U}$)

$\boldsymbol{U}^{f*}$= Common_U ($\boldsymbol{U}^f$)
$\boldsymbol{Q}^f$ = Face_Gradient($\boldsymbol{Q}, \boldsymbol{U}^f, \boldsymbol{U}^{f*}$)

$\boldsymbol{F}^{f*}$= Common_Flux($\boldsymbol{U}^f$)
$\boldsymbol{Q}$= Gradient_BR1($\boldsymbol{Q}, \boldsymbol{U}^f, \boldsymbol{U}^{f*}$)
$\boldsymbol{F}_v^f, \boldsymbol{R}$= Self_vFlux($\boldsymbol{U}, \boldsymbol{Q}, \boldsymbol{R}$)
$\boldsymbol{Q}^f$=BC_Gradient($\boldsymbol{Q}$)

$\boldsymbol{F}_v^{f*}$= Common_vFlux($\boldsymbol{U}^{f*}, \boldsymbol{Q}^f$)
$\boldsymbol{R}$= Correction($\boldsymbol{R}, \boldsymbol{F}^f, \boldsymbol{F}^{f*}, \boldsymbol{F}_v^f, \boldsymbol{F}_v^{f*}$)

図3 Runge-Kutta 1 ステージ内のFRアルゴリズムとデータ．$\boldsymbol{U}$は保存量から変換した基本変数，$\boldsymbol{Q}$はその勾配，$\boldsymbol{F}$は流束である．上付き文字の$\boldsymbol{f}$はセル境界面における物理量（左右2値ある）を表し，上付き文字の無いものはセル内のSPで定義された物理量である．赤字は粘性計算に必要な処理とデータを示す．

図4にGPU版LS-FLOW-HOのソースファイル環境のディレクトリ構成を示す．この図が示すようにGPU版LS-FLOW-HOを構成するソースファイルの内，OpenACCにてGPU化したソースファイルは別ディレクトリ（“openacc”ディレクトリ）配下に置かれており，CPU用ソースファイルと分けて管理をしている．また，GPU化したルーチンのルーチン名は末尾に“_OpenACC”を付加した名前としており（例：CPU用ルーチン“foo”に対応するGPU用ルーチン名

を“foo_OpenACC”としている），これらのルーチンの呼び出し制御はマクロ“_OPENACC”定義のプリプロセッサを用いて行っている．こうしたディレクトリ構造としている背景としては，GPU版LS-FLOW-HOのGPU化を進める際，GPU向けのソースファイル変更などを伴ったため，別ディレクトリおよび別ファイルでの管理を行ってきた．本来は同一ソースファイルにして指示行の切り替え（プリプロセッサ）をするのが理想と考えており，この点については開発効率や保守性を考慮しながら検討をしていきたい．

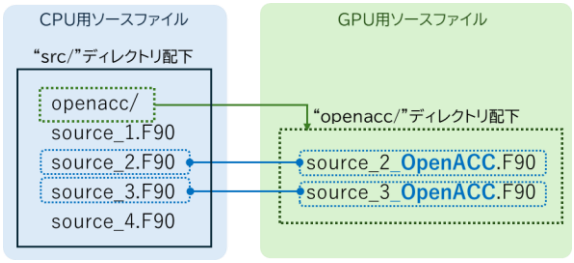

図4 CPU用ソースファイルとGPUソースファイルを配置しているディレクトリ構成．

４．OpenACC適用にあたっての課題と対策事例

4.1 OpenACC適用の基本方針

GPU版LS-FLOW-HOではOpenACCにてGPU化したソースファイルとCPU用ソースファイルが別ファイルであることから，現行バージョンへの適用は大きく下記のステップで行った．

① GPU版LS-FLOW-HOと現行版LS-FLOW-HOとのCPU部分のソースファイルの差分を確認

② 差分をGPU版LS-FLOW-HOのGPUソースファイルに適用し，現行版LS-FLOW-HOのGPUソースファイルとして組み込む

②に関しては，現行版LS-FLOW-HOにおけるGPU化対象ソースファイルに対してOpenACC化を適用することも検討したが，GPU化の経験が少なかったため，動作実績のあるGPUソースファイルに対して①で確認した差分を反映させていく進め方を採用した．

4.2 プログラムアボート発生に対する調査

②まで実施した時点でプログラムをGPU上で実行したところ，プログラムアボートが発生した．アボートの調査にあたっては，NVIDIA Compute Sanitizer[9]を使用して問題特定を行った．図5にCompute Sanitizerの起動例と出力例を示す．Compute Sanitizerを使用するにあたって，そのツール用にビルドする必要はなく，使用しているロードモジュールをそのまま使用することができる．Compute Sanitizerの情報は図5の「問題個所の出力」のような内容が標準出力に出力される．この例では“foo_OpenACC.F90”内のルーチン“foo_OpenACC”において2598行目で配列外参照(“out of bound”)が発生していることを示している．この例においては実際に図中の配列“array_B”に該当する配列において配列外参照が発生しており，当該配列の大きさの変更に伴う対応がなされていなかったことが要因であった．当該配列に対し適切な大きさを指定することにより本アボートを解消することができた．

Compute Sanitizerの起動

```
mpirun -n 4 --npernode 4 compute-sanitizer --tool memcheck --target-processes all ./a.out
```

問題個所の出力

```
========= Invalid __global__ write of size 8 bytes
=========     at foo_openacc_2328+0x18da0 in /data/LSHO_OpenACC/src/openacc/foo_OpenACC.F90:2598
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x50948d8 is out of bounds
=========     and is 22,736,966,943,528 bytes before the nearest allocation at 0x14ade1dfcc00 of size 13,312 bytes
```

該当箇所（サブルーチン“foo”: foo_OpenACC.F90）

```
2230    subroutine foo_OpenACC(arg)
(途中省略)
2596
2597          array_A(1,i) = T
2598          array_B(i)   = array_C(n,i)
2599
2600          array_D(i)   = Y
2601
2602          array_A(2,i) = P
2603          array_A(3,i) = Q
2604
2605       enddo
2606
2607       !$acc end parallel
```

図5 Compute Sanitizerの利用例．Compute Sanitizerの出力結果からアボートが発生している個所のソースファイルと行番号を知ることができる．

4.3 結果不一致の調査：演算部分の確認

アボート解消後，結果検証を実施した．GPU化における結果検証はNVIDIAのFortranコンパイラ“nvfortran”を使用してCPU実行用にビルドした現行版LS-FLOW-HOの計算結果と比較することにより行った．ここで得られた計算結果を以降，リファレンス結果と記す．結果検証を行った結果，リファレンス結果とGPU化した現行版LS-FLOW-HOの数値に大きな差異が発生していることが判明した．そこで結果不一致の調査を実施した．

最初に，GPUソースファイルへの演算部分の差分が正しく反映できているかどうかの確認を行った．この確認にあたっては，GPU版ソースコードに対しコンパイラオプションに“-noacc”を指定してOpenACC指示行を無効にし，かつマクロ“_OPENACC”を明示的に未定義にした状態でプログラムを生成した．このようにした目的は，GPUソースファイルを反映した現行版LS-FLOW-HOをCPU上で実行することであるが，“-noacc”を指定するとマクロ“_OPENACC”が定義されてしまいGPU上で実行してしまうため，このマクロを明示的に未定義にした．このようにしてビルドしたプログラムをCPU上で実行して得られた計算結果とリファレンス結果との比較を行った結果，計算結果の有意な差異はみられなかったことから，GPUソースファイルへの演算部分の差分が正しく反映できていることが確認できた．この結果から，結果不一致の要因はホストーデバイス間のデータ転送管理に問題があると想定した．“nvfortran”を使用してビルドしたOpenACCによるGPUプログラムのデフォルトのメモリモードはSeparate Memoryモードになる．Separate Memoryモードの場合，ホスト側のメモリとデバイス側のメモリを意識してデータ転送の管理をプログラマ責任で行う必要がある．そのため，データ転送管理が適切になされていないことが要因ではないかと考えた．

4.4 結果不一致の調査：Managed Memoryモードによる実行確認

データ転送管理に問題があることを確認するために，まずManaged Memoryモード[10]を使用してGPU実行を行い，結果検証を行った．Managed Memoryモードはホストーデバイス間のデータ転送管理をOpenACCの指示行に影響を受けることなくCUDAライブラリによって自動的に行う仕組みであり，コンパイラオプションの指定のみで使用できるようになる．この仕組みによりプログラマはホストーデバイス間のデータ転送を意識することなくGPU化を進めることが可能となり，GPUによる並列化に専念することができる．ただし，Managed Memoryモードではヒープ領域上に動的に確保した配列がデータ管理の対象となっているため，静的に確保した配列やスタック領域上の配列は管理対象外となることから，すべてのデータが対象となっていない点に注意が必要である．すべてのデータを管理対象とするためにはUnified Memoryモードを使用する必要がある[11]が，図6にある“nvaccelinf”コマンドによる当GPU環境情報の“Unified Memory”欄が示すように当方の環境ではUnified Memoryモードを利用できる環境ではなかったため，上記の点に留意しながらManaged Memoryモードを用いた．

Managed Memoryモードを使用したGPUコードでの結果検証の結果，有意な結果の差異は発生していなかった．このことから，コンパイラによるGPU化カーネル内の演算部分に対する計算結果への影響はなく，結果不一致の要因はホストーデバイス間のデータ転送管理に問題があることと断定することができた．

```
$ nvaccelinfo

CUDA Driver Version:           12020
NVRM version:                  NVIDIA UNIX x86_64 Kernel Module  535.129.03  Thu Oct 19 18:56:32 UTC 2023

Device Number:                 0
Device Name:                   Tesla V100-SXM2-32GB
Device Revision Number:        7.0
（途中省略）
Managed Memory:                Yes
Concurrent Managed Memory:     Yes
Preemption Supported:          Yes
Cooperative Launch:            Yes
Unified Memory:                No
Memory Models Flags:           -gpu=mem:separate, -gpu=mem:managed
Default Target:                cc70
（以下省略）
$
```

図6 nvaccelinfoコマンドの出力結果による当GPU環境の情報．“Managed Memory”欄が“Yes”，“Unified Memory”欄が“No”になっていることが分かる．

4.5 結果不一致の調査：データ転送管理の確認と対処

次にホストーデバイス間のデータ転送管理に問題があることを確認するために，OpenACC指示行の内，データ管理に関する指示行をすべて一旦外したGPUプログラム（GPUカーネル化の指定に関するOpenACC指示行のみ挿入された現行版LS-FLOW-HO）を生成した．データ管理に関する指示行がない場合，コンパイラはホストーデバイス間のデータ転送処理を行う処理をGPUカーネルの前後に組み込んだコードを生成する．まずはこのプログラムをSeparate Memoryモードで動作させて計算結果が正常であることを確認した．Separate Memoryモードで正常に動作するGPUコードを作成することができたので，外したデータ管理に関するOpenACC指示行を処理の先頭から順番に適用していった．

当初の想定ではこの手順を踏むことにより，適切なデータ転送管理を指定できていないOpenACC指示行を特定できる見通しであったが，外したすべてのOpenACC指示行を適用すると正常に動作をしてしまった．このことは，データ管理に関する指示行を外す前のソースファイルと今回適用したソースファイルに何らかの相違があり，その相違点が結果不一致を発生させていたと考えられる．本OpenACC適用対応はGitでバージョン管理を行っていたため，結果が不一致の状態のソースコードと今回のソースコードの差分を確認し，問題個所を特定することができた．

差分を確認したところ，デバイス側で使用する配列の初期化をホスト側で行っているが，デバイス側に初期化したデータを転送していなかった個所が見つかった．正しい値が初期値として設定されていない状態でデバイス側にて計算が行われていたため，正しく計算ができていなかった．この配列は３章で述べたようにメインループに入る前にデバイス上のメモリに確保されていることからホスト側ではなくデバイス側で初期化を行うようにした．こうすることにより，ホストーデバイス間のデータ転送を追加することなくGPUコードで正常に計算を行えるようになった．

4.6 OpenACCによるGPU化のまとめ

本章ではOpenACCを用いて現行版LS-FLOW-HOのGPU化を進めるにあたって遭遇した課題とその対処について述べた．実行時アボートの調査に関してはNVIDIA Compute Sanitizerを利用することにより，比較的時間を要することなくアボートの要因となっている個所を特定し解消することができた．一方，計算結果の不一致に関する調査にはある程度の時間を要してしまったが，本調査を進める上でたどった手順を図7にフローチャートとしてまとめた．フローチャート上の処理の内，今回の調査では通ることがなかった部分もあるが，今後のGPU化の取り組みにおいて該当するケースに遭遇した場合には，それらに関連する知見も蓄積していきたい．また，今回の不具合特定に至った結果から，GPU化に限らず，アボートや計算結果不一致などの調査をするにあたってはGit等のバージョン管理ツールを使用してソースファイルの履歴のトレースを容易にできる環境で行うことが望ましいと言える．

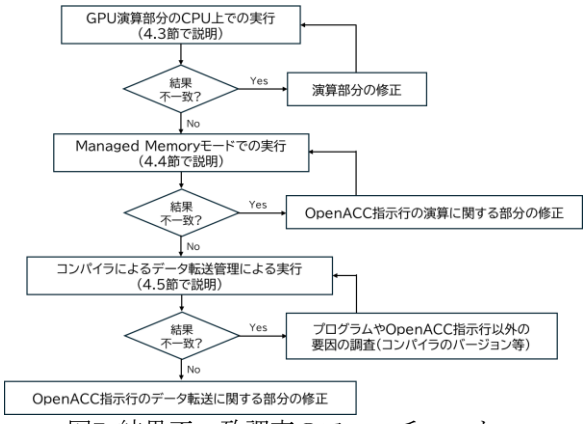

図7 結果不一致調査のフローチャート．

５．GPU化した現行版LS-FLOW-HOの性能状況

5.1 計算対象および評価条件

正常に計算を行えるようになった状態での性能評価は，２次元LOX/H2せん断層を計算対象として実施した．本計算対象の計算条件を表1に示す．また，図8に本計算対象の格子形状，図9，図10，図11にそれぞれ計算対象の密度分布，温度分布，圧力分布を示す．なお，図9～図11に示す分布は近似次数1の場合の計算を250,000ステップまで実行して得られた結果である．

表1 使用した計算対象の評価条件

総セル数	7,100
近似次数	2
燃焼計算	あり，FPVモデル
NS方程式の成分数	7
化学種の数	8
反応式の数	32
ルンゲクッタの段数	2
自由度	2,683,800
メインループのステップ数	1,000

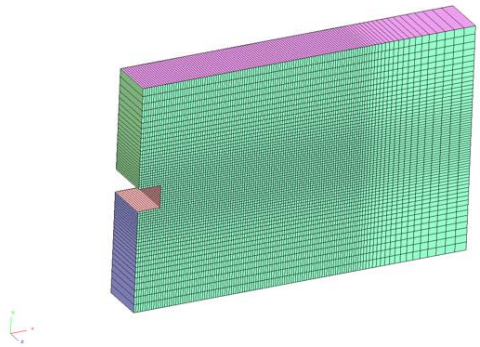

図8 計算対象の格子形状（セル数：7,100）．

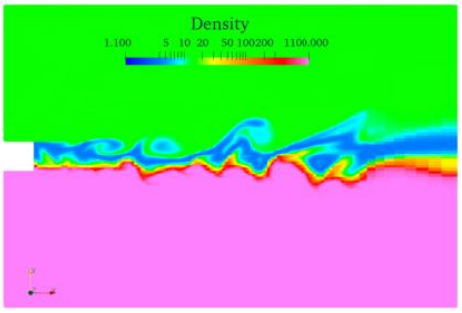

図9 計算対象の計算ステップ250,000時点の密度分布（次数：1の場合）．

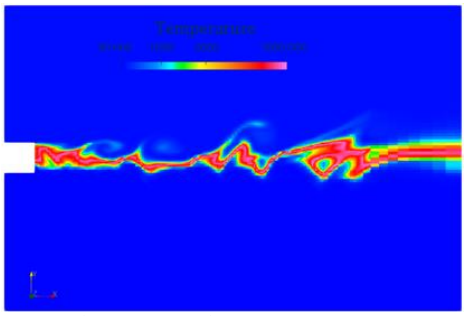

図10 計算対象の計算ステップ250,000時点の温度分布（次数：1の場合）．

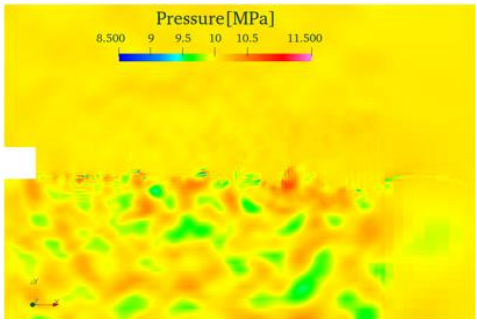

図11 計算対象の計算ステップ250,000時点の圧力分布（次数：1の場合）．

5.2 性能状況

表2に性能評価に用いた計算システムの諸元，表3に評価に使用したコンパイラの種類とバージョンを示す．

図12は今回性能評価に使用した計算機の理論演算性能，図13はそれぞれの計算機で実行した実行時間，図14はFX1000を1とした場合の実行演算性能比，図15はそれぞれの計算機で実行して得られた実行効率を示している．なお，図14の比較に用いた実行演算性能はFX1000で採取した浮動小数点演算量から算出した値で比較をした．

図12が示すようにFX1000 1ノードとV100 1カードの理論演算(倍精度実数演算)性能比較ではFX1000に対してV100が2.3倍高い値ではあるが，図14の実行演算性能比較をみると1.6倍となっている．1.6倍は理論演算性能比に及ばないが，空力版LS-FLOW-HOにおける実績[17]を考えると，性能向上の余地はあると思われることから，より詳細な分析を行った．

表2 性能評価対象システムのノード性能諸元

	FUJITSU PRIMEHPC FX1000[12]	Intel Xeon Gold 6240[13,14]	NVIDIA Tesla V100 SMX2[15,16]
コア数	48	36 (=18×2CPU)	2,560 (*2)
理論演算性能 (TFLOPS)	3.3792	2.9952	7.8
メモリ種別	HBM2	DDR4	HBM2
メモリバンド幅 (GB/s)	1,024	281.5	900
B/F (*1)	0.30	0.09	0.12
メモリサイズ (GB)	32	384	32
キャッシュ (MB)	32	49.5 (=24.75×2CPU)	6

(*1) B/Fはメモリバンド幅をピーク性能で除した値，(*2) GPUあたりのFP64コア数

表3 評価に使用したコンパイラ

評価システム	使用コンパイラ
FUJITSU PRIMEHPC FX1000	frtpx (FRT) 4.11.2 20240524
Intel Xeon Gold 6240	ifx (IFX) 2025.0.0 20241008
NVIDIA Tesla V100 SMX2	nvfortran 24.11-0 64-bit target on x86-64 Linux -tp cascadelake

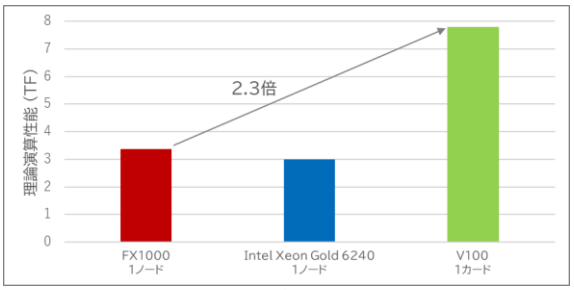

図12 理論演算性能比較.

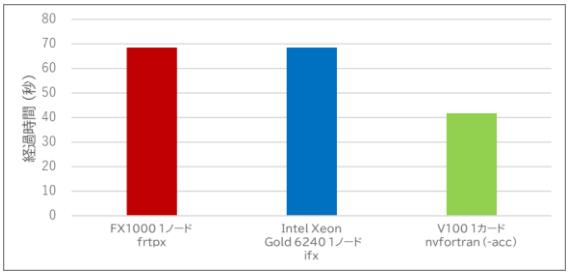

図13 計測区間時間比較.

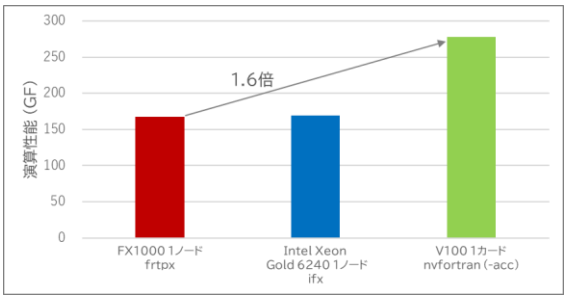

図14 FX1000=1とした場合の実行演算性能比.

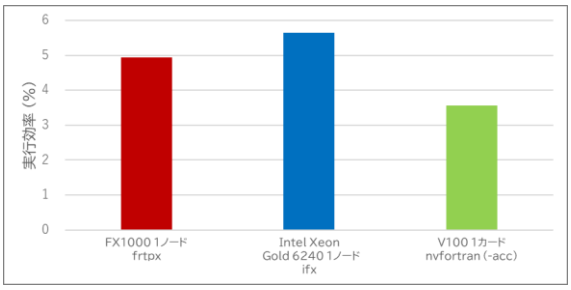

図15 各評価システムにおける実行効率比較.

まずは，NVIDIAの性能分析ツールのひとつであるNsight Systems[18]を使用してプログラムの実行全体を俯瞰した（図16）．この図は計測区間全体のタイムライン情報を示しており，タイムラインの左側に緑色，右側に赤色で表示されていることが分かる．緑色の部分が主にホストからデバイスにデータ転送を行っている部分であり，赤色の部分がデバイスからホストにデータ転送を行っている部分である．この図から計測区間の前後で主にデータ転送が発生していることを示している．図17はメインループの1ステップあたりのタイムライン情報を示したものであり，ホストーデバイス間のデータ転送がほとんど発生していないことが分かる．

図18はタイムライン情報の左側に表示されている情報を拡大した図である．この図をみると“CUDA HW”の個所に“98.7% Kernels”と“1.3% Memory”という表示があり，Memoryの個所にはHtoD memoryとDtoH memoryという表示がある．この割合はCUDA HWで実行された時間の内，98.7%が演算処理，1.3%がホストーデバイス間のデータ転送を含むメモリ関連の処理に要していることを示しており，この比率から本プログラムにおいてはホストーデバイス間のデータ転送時間が全体に占める割合が少ないことが分かる．これは３章で説明したメインループに入る前に計算に必要な配列をすべてデバイスのメモリ上に確保してデバイス上で計算を行っているため，ホストーデバイス間のデータ転送による影響の極小化を実現できていることを示している．またこの情報ではルーチンのコスト分布の情報も表示されており高コストルーチンの特定に役立つ．

ホストーデバイス間のデータ転送による影響が少ないことから，Nsight Compute[19]を使用してルーチン単位の性能状況を見てみた（図19）．この図は各サブルーチンの情報を実行時間(“Duration”欄の情報)で時間を要している順にソートしたものである．この図が示すように最も時間を要しているルーチンは“transprop_sp_openacc”である．ルーチン名の右端に表示されている番号はソースファイル上の行番号であり，この図では“transprop_sp_openacc”が記述されているソースファイルの546行目でGPUカーネル化された部分の情報を表示していることになる．このルーチンをクリックすると図20のようにNsight Computeが分析した性能情報が表示される．

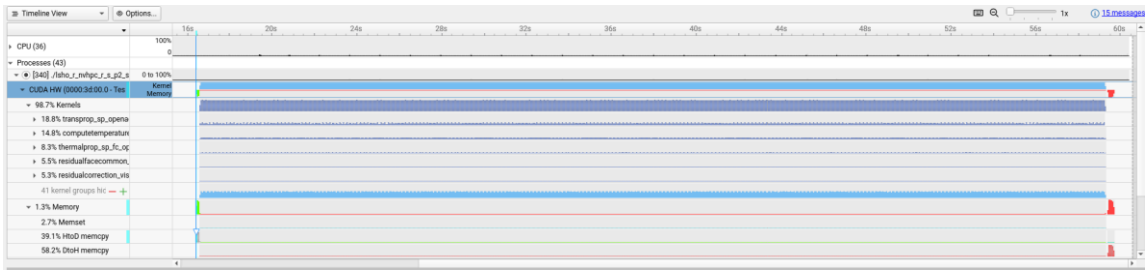

図16 Nsight Systemsの出力画面（上図が計測区間全体のタイムライン情報）.

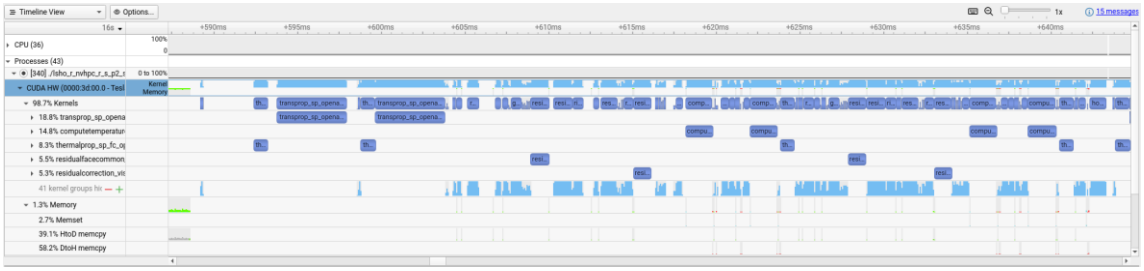

図17 Nsight Systemsの出力画面（1ステップあたりのタイムライン情報）．

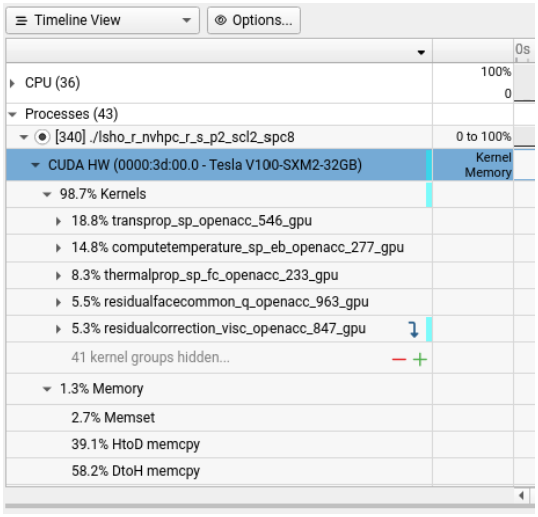

図18 Nsight Systemsによるコスト分布の情報．

この図ではSMSP(SM sub-partition), SM(Streaming Multiprocessor), L1(L1 キャッシュ)の3種類の分析結果が表示されている．この内，SMSPの診断メッセージをみてみると“One or more SMSPs have a much lower number of active cycles than the average number of active cycles. Maximum instance value is 34.07% above the average, while the minimum instance value is 99.91% below the average.”とあり，下記のことを指している．

・一部のSMSPは平均より34.07%多く稼働
・別のSMSPは平均より99.91%少なく稼働
（ほとんど稼働していない）

SMSPとはSMのサブパーティションのことであり，V100の場合，図21が示すようにSMあたり4つのサブパーティションで構成される．これらの情報はV100の演算資源を均等に利用できていないことに起因するロードインバランスが発生している可能性を示唆している．ロードインバランス発生の要因として計算規模が小さい点を挙げることができる．２章で述べたようにLS-FLOW-HOのGPU化は基本的にセル数のループに対して行われているが，本評価ケースでは表3が示すように総セル数が7,100と少ない．上記の分析結果から，現状で1.6倍の性能比は計算規模が小さいため演算資源を有効に活用できていないことによるものと考えられることから，妥当と見ている．今後は問題規模を拡大した性能分析を実施し，更なる性能向上を検討していく方針である．

６．おわりに

本稿では，旧バージョンで適用したOpenACCによるGPU化施策を現行バージョンに適用するという取り組みについて述べた．本取り組みあたっては，GPU化の経験の少ない状態で実施し，そこで得られた知見の共有を行った．GPU化にあたっては正常動作をするまでに幾つかの問題点を解決しなければならなかったが，その多くにおいてNVIDIAが提供するツールや機能を活用することにより対応することができた．このことから，これらのツールや機能を事前に理解しておくことにより，より効率的に調査を進めることができると思われる．我々を取り巻く計算環境ではGPUが主流になってきていることは容易に想像することができ，GPU化の経験が少ない開発者・研究者も自分たちのプログラムのGPU化に取り組みことが避けられない状況になりつつあると思われる．こうした状況において本稿の内容が少しでも参考になれば幸いである．

今回はOpenACCによるGPU化の経験を通じて幾つかの知見を得ることができた．今後はGPU環境においてプログラムの性能を引き出すための知見の蓄積も進めていきたい．

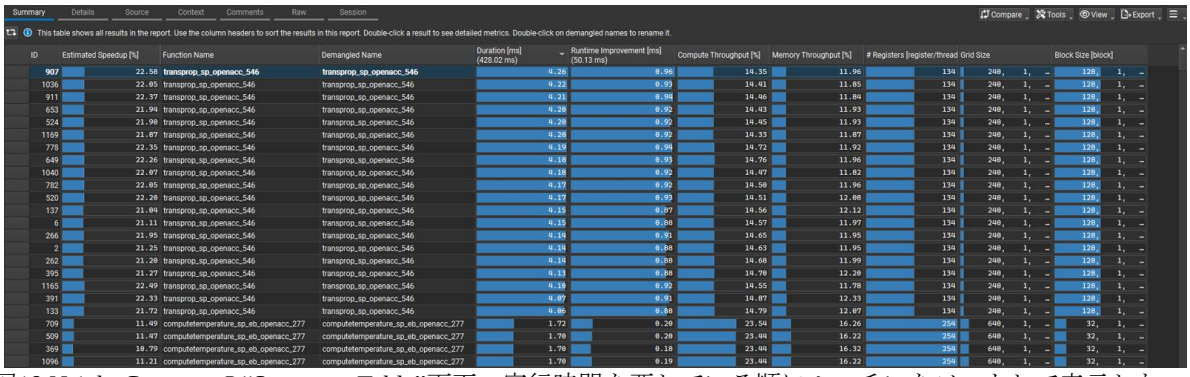

図19 Nsight Computeの“Summary Table”画面．実行時間を要している順にルーチンをソートして表示した．

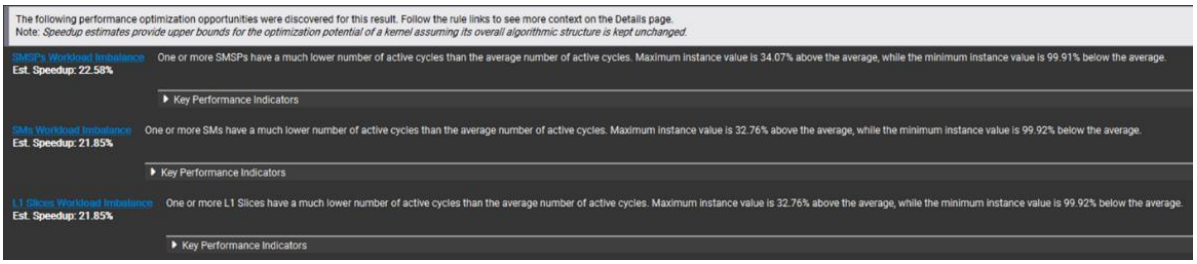

図20 Nsight Computeの“Prioritized Rules”画面．推定される潜在的な高速化に関する情報が出力される．

図21 Volta GV100のSM のブロック図[20]

謝辞

数値計算の実行には宇宙航空研究開発機構のスーパーコンピュータJSS3を使用した．ここに記して感謝の意を表す．

参考文献

[1] November 2024 | TOP500 https://www.top500.org/lists/top500/2024/11/
[2] JSS3: JAXA Supercomputer System Generation 3 https://www.jss.jaxa.jp/
[3] 芳賀，福島，熊畑，根岸，清水，“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 (208), 126-135 (2022).
[4] 芳賀，多湖，“GPUによる圧縮性燃焼ソルバLS-FLOW-HOの高速化,” 流体力学講演会／航空宇宙数値シミュレーション技術シンポジウム2023 3A04, (2023).
[5] Pierce, C., Moin, P., J. Fluid Mech 504: 73-97 (2004).
[6] Soave, G., Chemical Engineering Science, 27:1197-1203, 1972.
[7] Chung, T.H., Ajlan, M., Lee, L.L., and Starling, K.E., Ind. Eng. Chem. Res., 27:671 – 679, 1988.
[8] Huynh, H.T., AIAA Paper: 2007-4079 (2007).
[9] Compute Sanitizer https://docs.nvidia.com/compute-sanitizer/ComputeSanitizer/index.html
[10] GPUプログラミング入門 (OpenACC) https://www.hpc.cmc.osaka-u.ac.jp/wp-content/uploads/2024/06/20240627_OpenACC_Basic.pdf
[11] NVIDIA CUDA Fortran Programming Guide https://docs.nvidia.com/hpc-sdk/archive/24.11/compilers/cuda-fortran-prog-guide/index.html
[12] FUJITSU Supercomputer PRIMEHPC FX1000 https://www.fujitsu.com/downloads/JP/jsuper/primehpc-fx1000-datasheet-ja.pdf
[13] インテル® Xeon® Gold 6240 プロセッサー https://www.intel.co.jp/content/www/jp/ja/products/sku/192443/intel-xeon-gold-6240-processor-24-75m-cache-2-60-ghz/specifications.html

[14] 第3世代JAXAスパコンが目指すもの -"TOKI"導入の目的と初期性能評価-
https://www.jss.jaxa.jp/mediadir/2021/02/TOKI_ 概要.pdf
[15] NVIDIA TESLA V100 GPU アクセラレーター
https://images.nvidia.com/content/technologies/deep-learning/pdf/NVIDIA-Tesla-V100-JPN.pdf
[16] NVIDIA TESLA V100 GPU アーキテクチャ 世界最先端のデータセンター GPU
https://images.nvidia.com/content/pdf/tesla/Volta-Architecture-Whitepaper-v1.1-jp.pdf
[17] 渡部, 芳賀, 高木, 高次精度燃焼ソルバLS-FLOW-HOの複数計算アーキテクチャにおける性能分析およびベクトル演算機構による高速化,” 流体力学講演会／航空宇宙数値シミュレーション技術シンポジウム2024 3B08, (2024).
[18] NVIDIA Nsight Systems
https://developer.nvidia.com/nsight-systems
[19] NVIDIA Nsight Compute
https://developer.nvidia.com/nsight-compute
[20] NVIDIA TESLA V100 GPU ARCHITECTURE THE WORLD’S MOST ADVANCED DATA CENTER GPU
https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf