

GKV の移植・最適化報告

前山伸也

核融合科学研究所 メタ階層ダイナミクスユニット

プラズマシミュレーションポジウム2025, 核融合研, 2025年9月11日 9:30-9:55

Outline

- GKVコード概要
- サブシステムAでの性能評価
- サブシステムBでの性能評価
- まとめ

GyroKinetic Vlasov code (GKV)

- ジャイロ運動論モデルに基づく核融合炉心プラズマ乱流の第一原理シミュレーション
- 磁力線に沿ったフラックスチューブモデルを用いた高精度計算
- 5次元位相空間上の移流・拡散
 - MPI/OpenMPハイブリッド並列
 - FFTスペクトル法(x, y) + 差分法($z, v_{\parallel}, \mu$)
 - 陽的ルンゲクッタ法 + 陰解法衝突項

◆GKVの演算とMPI通信パターン

- z, v 差分： z, v 方向軸通信 $\left(n_x, \frac{n_y}{P_w}, \frac{n_z}{P_z}, \frac{n_v}{P_v}, \frac{n_{\mu}}{P_{\mu}}, \frac{n_s}{P_s}\right)$
- x, y 2次元FFT：転置通信 $\left(n_x, n_y, \frac{n_v}{P_v}, \frac{n_z n_{\mu}}{P_z P_{\mu} P_w}, \frac{n_s}{P_s}\right)$
- 密度計算： vms 総和通信
- 衝突項陰解法：転置通信 $\left(n_v, n_{\mu}, n_s, \frac{n_z}{P_z}, \frac{n_x n_y}{P_w P_v P_{\mu} P_s}\right)$

(転置後、 v, μ, s に関する総和・差分の反復計算)

$$\frac{\partial \tilde{f}_s}{\partial t} + \left(v_{\parallel} \frac{\mathbf{B} + \tilde{\mathbf{B}}_{\perp}}{B} + \mathbf{v}_{sG} + \mathbf{v}_{sC} + \tilde{\mathbf{v}}_E \right) \cdot \nabla \tilde{f}_s + \frac{dv_{\parallel}}{dt} \frac{\partial \tilde{f}_s}{\partial v_{\parallel}} = S_s + C_s,$$

$$\nabla_{\perp}^2 \tilde{\phi} = -\frac{1}{\epsilon_0} \sum_s e_s (\tilde{n}_s + \tilde{n}_{s, \text{pol}}),$$

$$\nabla_{\perp}^2 \tilde{A}_{\parallel} = -\mu_0 \sum_s e_s \tilde{u}_{\parallel s},$$



最近の物理モデル・数値実装拡張

➤ 速度空間座標系選択 ($v_{\parallel}, \mu$) $\leftrightarrow$ ($v_{\parallel}, v_{\perp}$)

磁場強度が空間的に大きく変化するダイポール磁場配位でも効率的に計算

➤ 大規模並列データ出力

従来① 各MPIランク毎にFortranバイナリ出力 → ユーザーが読み込みにくい

従来② NetCDF/Parallel HDF5による単一NetCDF出力 → 可搬性や並列書き出し性能に課題

新実装: Zarrフォーマット → JSONファイルによるメタデータ記述を加えるだけで外部パッケージに依存しない可搬性、各MPIランク毎にデータを書き出す完全並列性、ユーザー読み込みの利便性



- 大規模多次元配列データを格納するためのオープンソースフォーマット
- ディレクトリによるグループ分けとチャンク分割データの階層化管理
- 各階層にJSONメタデータ記述（チャンク構成、データ型、圧縮コーデック、...）
- クラウド対応（Amazon AWSやGoogle, Azureでも利用されている）、Daskなどの非同期並列処理にも対応しスケーラブル

➤ VMEC平衡読み込みモジュール

オープンソースプラットフォームへの公開による簡便なセットアップ `$ pip install bzx`

VMECによるMHD平衡構築 → BoozXformによる磁束座標構成 → BZXによるGKV座標入力を円滑に。

Outline

- GKVコード概要
- サブシステムAでの性能評価
- サブシステムBでの性能評価
- まとめ

サブシステムAのシステム構成

モデル名：NEC LX 204Bin-3

標準的なCPUシステムだが、CPU当たり
コア数が128と多いこととNUMA構成に注意。
ノード当たりメモリ容量768 GB

並列度：

- 360 ノード
- 2 CPU/ノード (Intel Xeon 6980P; Granite Rapids)
- 3 NUMA/CPU
- 42 core/NUMAが1つと43core/NUMAが2つ

CPU性能（＝ノード性能÷2）：

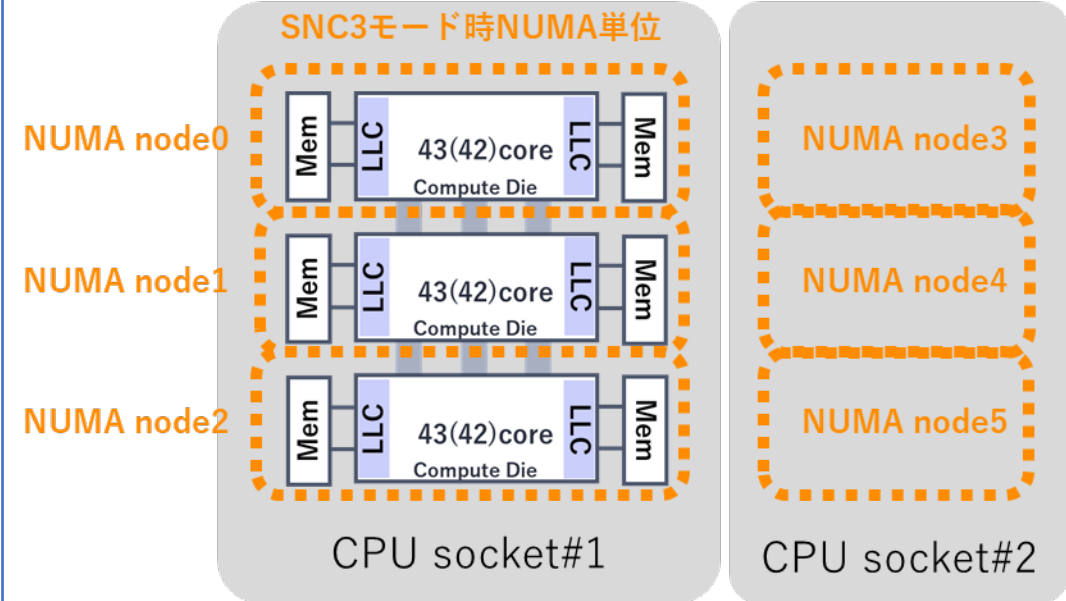
- CPU演算性能(倍精度): 8.19 TFLOPS
- CPUメモリバンド幅: 0.844 TB/s

参考：Fugaku CPU（＝ノード）性能: 48 cores, 3.072 TFLOPS, 1.02 TB/s

SX-Aurora VE性能: 8 cores, 2.43 TFLOPS, 1.35 TB/s



2 CPU/ノード

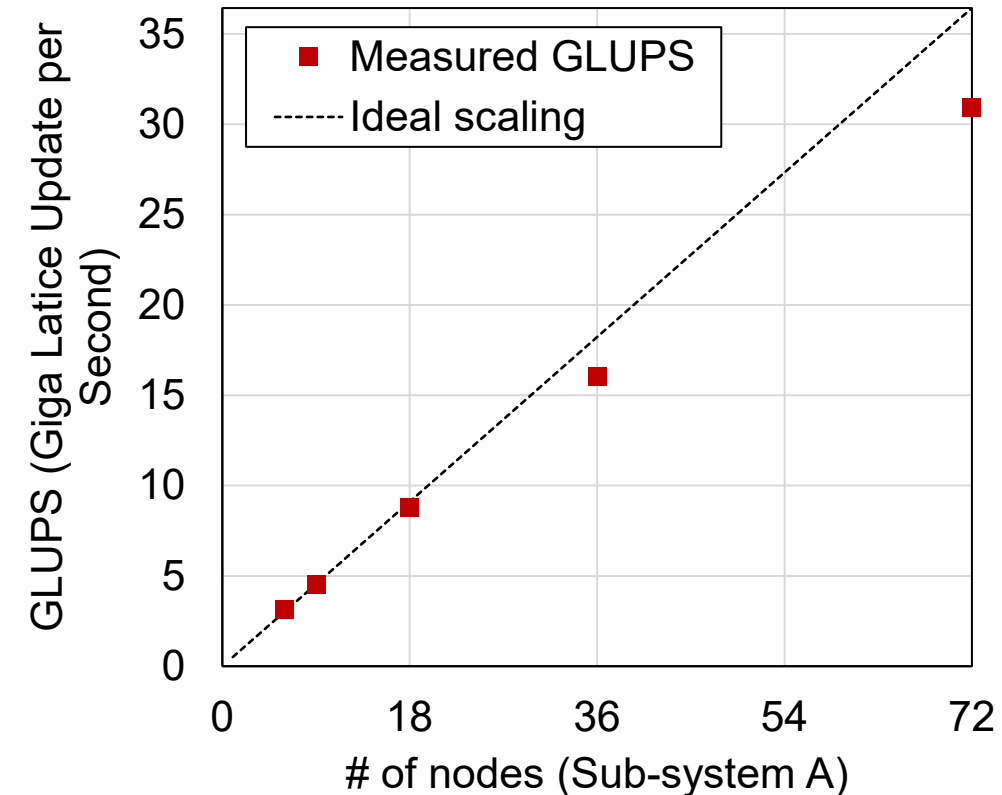


サブシステムAでのGKV性能評価設定

問題設定： e, D^+, C^{+6} 多粒子種衝突項を含むイオンスケール乱流シミュレーション

電子-不純物衝突による時間刻み幅律速を避けるため、衝突項陰解法反復計算が主要部
富岳での性能評価では理論ピーク演算性能比8~10%程度 of 良好な性能を達成

- 解像度: $(N_x, N_y, N_z, N_v, N_\mu, N_s) = (256, 256, 24, 120, 48, 3) = 2.7 \times 10^{10}$ 格子点
- MPI分割数 $(P_w, P_z, P_v, P_\mu, P_s) = (4, 3, 6, 4, 3) = 864$ プロセス
- OpenMP/MPI = 10 スレッド
- 各NUMAに4 MPI割り当て(40スレッド/42core)
- 利用ノード数36 (良好なストロングスケール
ング範囲内)



サブシステムAでのGKV性能評価結果

- Sub-system Aはメモリバンド幅で劣るが、FugakuやSX-Auroraより高い理論ピーク性能比
- (主)衝突項陰解法の高キャッシュ利用率（大容量L3キャッシュ504MB）、(副)ノード数が少ないことによる通信コスト削減、が効いていると考えられる。

→ サブシステムAでは高い演算性能を達成

評価システム	Sub-system A	SX-Aurora	Fugaku
利用ノード数 (総理論演算性能)	36 (589.68 TFLOPS)	480 (1164 TFLOPS)	384 (1180 TFLOPS)
CPU理論演算性能	8.19 TFLOPS	2.43 TFLOPS	3.072 TFLOPS
CPU理論メモリバンド幅	0.844 TB/s	1.35 TB/s	1.02 TB/s
経過時間/20ステップ	51.84 sec	35.41 sec	42.64 sec
内訳：ポテンシャル計算	4.43 sec	0.74 sec	6.12 sec
内訳：線形項計算	11.11 sec	4.56 sec	7.89 sec
内訳：非線形項計算	7.22 sec	3.95 sec	4.27 sec
内訳：衝突項陰解法計算	21.05 sec	20.77 sec	18.76 sec
総理論演算性能当たり処理速度（対Fugaku比）	1.65 倍	1.22 倍	1

Outline

- GKVコード概要
- サブシステムAでの性能評価
- **サブシステムBでの性能評価**
- まとめ

サブシステムBのシステム構成

モデル名：NEC LX 401Bax-3GA

AMDのUnifiedメモリ(CPUとGPUでメモリ空間を共有)

対応APUシステム

ノード当たりメモリ容量512 GB

並列度：

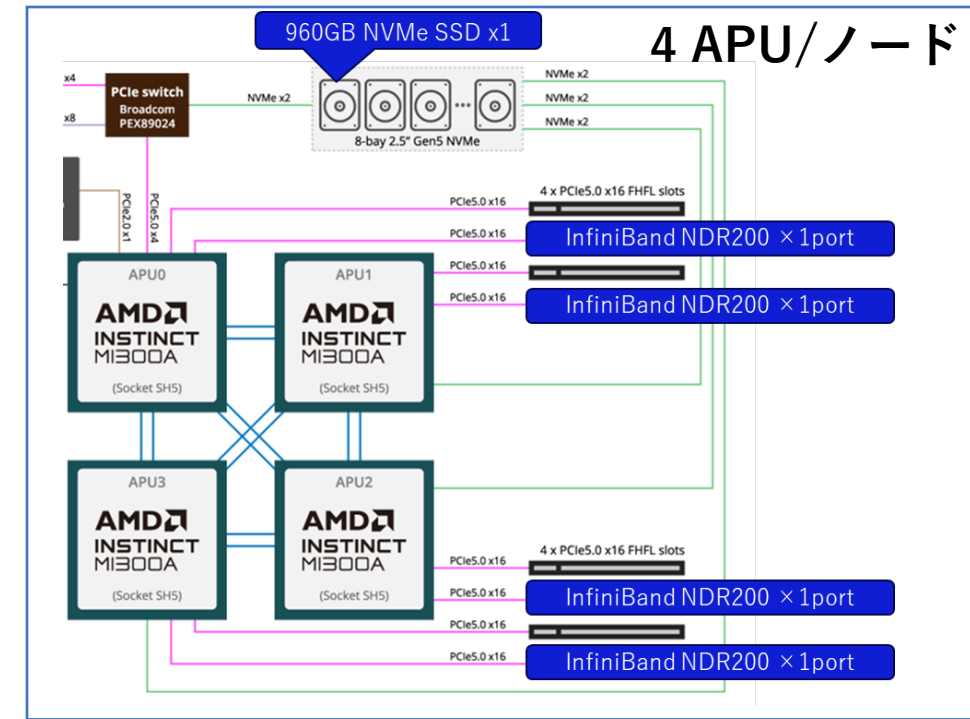
- 70 ノード
- 4 APU/ノード (AMD Instinct MI300A)
- 228 CU/APU (CU: Computing Unit)
- 64 StreamCore/CU

→ APUの演算性能を発揮させるには、14,592個の演算コア(Stream core)を如何に効率的に動作させるかが重要

APU性能 (= ノード性能 ÷ 4)：

- APU演算性能(倍精度ベクトル演算性能): 61.3 TFLOPS
- APUメモリバンド幅: 5.3 TB/s

※倍精度マトリックス演算性能122.6 TFLOPS



GKVのGPU環境対応

GPU対応の実装方法：Fortran + OpenMP target

CPUにおけるOpenMP実装と用語のイメージが若干ずれるので注意

- CPUの場合、大規模ループをOpenMP threadsで分割し、物理コアに対応する様に割りあてる。
- GPUの場合、大規模ループをOpenMP teamsで分割し、ComputingUnitへ順次割り当て。各team内ではさらにOpenMP threadsで分割し、StreamCoreへ順次割り当て。

ハードウェア: GPU AMD Instinct MI300A	OpenMPプログラミングとの対応	ハードウェア: CPU Intel Xeon 6980P	OpenMPプログラミングとの対応
ComputingUnit数: 228	OpenMP teams	物理コア数: 128 (※ただし3 NUMA構成)	OpenMP threads
StreamCore数/CU: 64 (1クロックあたりに実行可能な演算器の総数14,592)	OpenMP threads	AVX-512数/cores: 2 (1クロックあたりに実行可能な演算器の総数256)	
命令列単位: 64論理スレッド(=1 warpと呼ぶ)		命令列単位: 1論理スレッド(AVX-512利用時は512bit SIMD)	simd指示文

※2種類の「スレッド」という用語の使い分け
ハードウェアスレッド（論理スレッド）・・・CPUが同時に実行可能な命令列の単位
ソフトウェアスレッド（OpenMPスレッドなど）・・・ユーザー空間で生成される並列処理の単位

参考：GPU処理イメージ（サブシステムB向け）

```
!$OMP target teams distribute collapse(2)
do m = 1, n5
  do l = 1, n4
    !$OMP parallel do collapse(3)
    do k = 1, n3
      do j = 1, n2
        do i = 1, n1
          (...処理...)
```

- 外側2重ループをcollapseさせて、target teams distributeでOpenMP Team作成
→ ComputingUnit向け論理ブロック。Team数はCU数=228の数倍(~6倍)程度。この際、collapseさせたループ総数よりTeam数が少なくても許容。Team数分の計算実行を複数回行い、全ループを処理する。
- 残りの内側3重ループをcollapseさせて、parallel doでOpenMP Thread作成
→ StreamCore向け論理スレッド。スレッド数はSC数=64の数倍(~4倍)程度。

※あくまで処理イメージ。実際には、!\$omp target teams distribute parallel do collapse(5)とまとめて、team数およびスレッド数を自動割り当てする形で実装。

参考：CPU処理イメージ（サブシステムA向け）

```
!$OMP parallel do collapse(3)
  do m = 1, n5
    do l = 1, n4
      do k = 1, n3
        do j = 1, n2
          !$OMP simd
            do i = 1, n1
              (...処理...)
```

- ・ 外側ループをcollapseさせて、parallel do
→ CPUでは典型的にOpenMPスレッドを物理コアに対応するように割り当てる。
上記の単純なループではchunkサイズはstaticに自動分割される最大粒度で十分と思われるが、一般に複雑なループ処理の場合はchunkサイズもある程度まとまった粒度で適切に指定する方がよい。
- ・ 最内ループはSIMD化

※GPUの場合は最小単位であるところのthreadで最内ループまでつぶす必要があるのに対し、CPUの場合はthreadは物理コアに渡すまとまった単位であり、最内ループはSIMD用に残す必要があるという違い。
→ポータブルなコードにするには、単一の実装ではなく、同一の5重ループに対して、GPU向けOpenMP指示行を加えた記法と、CPU向けOpenMP指示行を加えた記法をそれぞれ併記し、マクロで切り替えるような実装が現実的。

サブシステムBでのGKV性能評価結果

- 現状：NECによる移植作業がひとまず完了。**サブシステムBでシミュレーション完遂。**
- 衝突項陰解法モジュールは、Fortranでコンパイルが通らずC言語化。ループ構造変更。
 - 非線形項モジュールは、1次元多重FFTに置換した上でHIP FFT (AMD GPU最適化FFT)を使用。
 - malloc/allocateで取っていた配列をローカル配列化。

- ただし、性能が十分に出ているとはいいがたく、作業継続中。
- SX-Aurora → サブシステムA: 演算性能3.3倍、メモリ性能0.6倍に対し、実行速度2.0倍
 - SX-Aurora → サブシステムB: 演算性能25.2倍、メモリ性能3.9倍に対し、実行速度1.27倍

実行環境		ノード数 VE 数	ソケット数 APU数 VE数	プロセス数	プロセス分割	スレッド数	ブロックサイズ	Timesteploop (sec)	各システムの理論 演算・メモリ性能
サブシステムB (MI300A)	CPU実行	3	12	12	(4,1,1,1,3)	24	-	1448.6	61.3 TFLOPS/APU 5.3 TB/s/APU
	CPU+GPU実行	3	12	12	(4,1,1,1,3)	24	1197	566.3	
サブシステムA (Intel Xeon 6980P)	CPU実行	6	12	1296	(4,6,6,3,3)	flat	-	359.3	8.19 TFLOPS/CPU 0.844 TB/s/CPU
※参考 SX-Aurora TSUBASA (Type10AE)	VE実行	48	48	384	(4,2,4,4,3)	flat	-	179.6 (12VE換算:718.4)	2.43 TFLOPS/VE 1.35 TB/s/VE

Outline

- GKVコード概要
- サブシステムAでの性能評価
- サブシステムBでの性能評価
 - 追加：その他コードによる単体性能ベンチマーク
- まとめ

単体性能ベンチマーク① HIPコードとの比較

3次元拡散方程式（8次精度中心差分） $\frac{\partial f}{\partial t} = \nu \nabla^2 f$

格子点数：512×512×512

目的：サブシステムBにおけるFortran + OpenMPの有効性を検証するため、HIP実装と比較。
単一APUでの実行。

評価システム	Sub-system B HIP実装 [thanks to 伊藤さん]	Sub-system B Fortran+OpenMP (target)	Sub-system A Fortran+OpenMP (threads)
演算性能	61.3 TFLOPS/APU	61.3 TFLOPS/APU	8.19 TFLOPS/CPU
メモリバンド幅	5.3 TB/s/APU	5.3 TB/s/APU	0.844 TB/s/CPU
経過時間/10ステップ	0.056 sec	0.717 sec	0.463 sec
経過時間/100ステップ	0.546 sec	1.308 sec	4.212 sec
経過時間/1000ステップ	5.459 sec	6.926 sec	(21.373 sec)

- HIPはSub-system Aと比較して4.212/0.546=7.7倍高速であり、妥当。
- **Fortran+OpenMP(target)によるAPU利用も、暗黙の初期化コストが載っているようだが、ステップ数が多くなればHIP実装に漸近しており、有効性が認められた。**

単体性能ベンチマーク② PythonによるAPU利用

Pythonでの数値演算もGPU利用可能なパッケージを利用することで高速化可能。

インストール方法：“利用の手引き7.7 ROCm 対応のpipインストール”を参考に、各パッケージに応じて適切にインストールする。

1. pyenvを利用してROCm 6.3.3対応のPython 3.12.11をインストール
2. サブシステムBにログインし、ROCm対応パッケージをインストール

```
module load rocm
```

```
pip install https://repo.radeon.com/rocm/manylinux/rocm-rel-  
6.3.3/jax_rocm60_pjrt-0.4.31-py3-none-manylinux_2_28_x86_64.whl --  
proxy http://10.20.64.14:8080
```

(...同様にjax_rocm60_plugin, jaxlib, jaxなどもインストール...)

単体性能ベンチマーク② PythonによるAPU利用

シロアリ塚形成シミュレーション

(3次元反応拡散方程式 $\frac{\partial f}{\partial t} = -f(1-f)\nabla^2 f - \nabla^4 f$)

格子点数: $128 \times 128 \times 128$

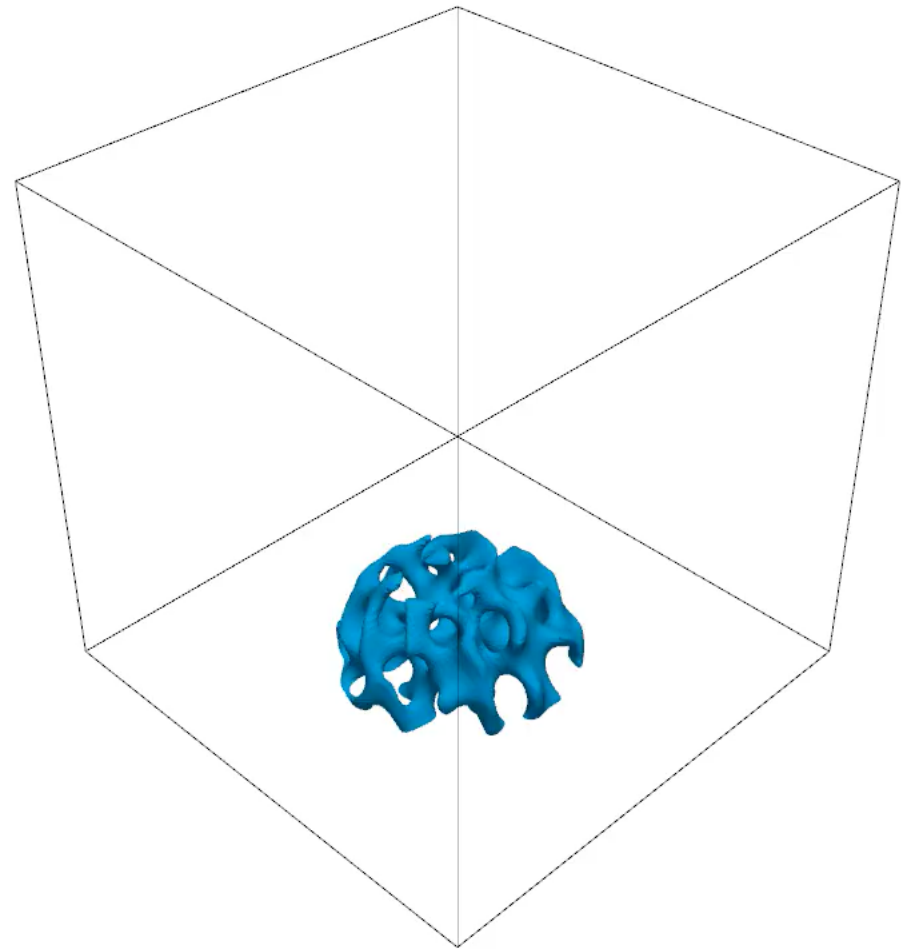
空間差分: 2次精度中心差分

時間積分: 作用素分割 + 2次精度陽的Runge-Kutta法 +
1次精度陰的Euler法

jaxによる純粋Pythonコードとして実装し、サブシステムBで計算実行。

- CPU利用時経過時間: 456秒
- **APU利用時経過時間: 19秒**

→ 対応パッケージのインストールとサブシステムB上でのジョブ実行により、PythonでのAPU利用による計算の高速化ができることを確認。



まとめ

- ✓ GKVコードは継続的に開発。
 - 物理モデル拡張（ダイポール磁場、電離層-磁気圏結合系）
 - Zarrフォーマット出力形式
 - MHD平衡インターフェースのオープンPythonパッケージ化
- ✓ サブシステムAでは、富岳やSX-Auroraと比べて高い理論ピーク性能比を達成
 - 非一様メモリアクセス（NUMA）構成に合わせたノード内ハイブリッド並列
 - **大規模L3キャッシュの再利用効率を高めることでメモリバンド幅克服**
 - 特に、5次元大規模問題を3次元の小規模問題（=オンキャッシュ）に分割して、反復法（=再利用）で解く、という今回の問題設定が上手くハマった。
- ✓ サブシステムBでの、Fortran+OpenMP(target)によるGKVコード実行を完遂。
 - 一部はC言語化して対応。AMD製Fortranコンパイラの成熟が待たれる。
 - SX-AuroraやサブシステムAと比較して、実行性能が不十分であり、さらに最適化を継続。