修士論文

GPGPU を用いた決定論的手法に基づく α固有値計算の高速化

名古屋大学大学院

工学研究科博士前期課程

総合エネルギー工学専攻

山本章夫研究室

山口 響

令和 6 年 2 月

GPGPU を用いた決定論的手法に基づくα固有値計算の高速化

山本章夫研究室　山口 響

1. 緒言

即発中性子減衰定数αとは、体系内に存在する中性子数が1/e倍に減少するまでの時定数の逆数に対応する核特性である。近年、中性子非増倍体系でも測定可能な即発中性子減衰定数αを、数値解析結果の妥当性確認や、データ同化による評価済み核データ更新に活用する手法が模索されている。ただし、決定論的手法に基づいた即発中性子減衰定数αの固有値計算を考えた場合、輸送計算の反復法に多くの計算時間を要するため、少ない角度分割数で済む高精度な角度分点や、拡散加速法の実装および反復計算の高速化が必要となる。また、水素原子核を多く含む水槽体系のように、非等方散乱が核特性に影響を及ぼす体系では、k_{eff}計算と同様に非等方散乱中性子源を精度良く取り扱う必要もある。以上の課題を解決するため、本研究では GPGPU を利用した高速な計算が可能な、非等方散乱中性子源を考慮したS_N法によるα固有値計算コードを開発した。

2. GPGPU を用いたα固有値計算コードの実装

GPGPU とは、画像処理装置である GPU を数値計算等に利用する技術であり、様々な物理シミュレーションや機械学習など広い分野で活用が進んでいる。本研究では GPGPU による並列計算向けプラットフォームの 1 つである CUDA を利用して、特に計算コストの高い①角度中性子束の更新を行う transport sweep、②非等方散乱中性子源の更新、③拡散加速計算、以上 3 つの処理を改善した計算コードを新たに開発した。

GPU の性能を活用するには大量のデータに対して並列に処理する必要があるため、transport sweep の実装では、エネルギー、中性子飛行方向、空間メッシュについて計算を並列化した。ここで、transport sweep を飛行方向について並列化する場合、高次の非等方散乱を取り扱うために必要な各次の中性子束モーメントϕ_m^lを計算する際にスレッド同士で操作が競合する。そのため、高い実行コストを伴う不可分操作が多数回必要であり、演算性能低下が課題となり得る。そこで本研究では、shared memory(GPU の小容量・高速なメモリ)を利用したうえで、各スレッドが計算する展開次数の組(l,m)をずらして競合を回避することで、必要な不可分操作の回数の削減を図った。これにより改善前の transport sweep コードに比べ約 2.8 倍の性能が達成できた。

3. 計算結果

開発したコードの有効性および妥当性を確認するため、過去実験[1]の水槽体系について非等方散乱を P7 成分まで考慮したα固有値計算を実施した。最大寸法の水槽体系について、エネルギー172 群、飛行方向分割数 1200、空間メッシュ数36^3の条件下で計算を実施したときの CPU・GPU コードによる計算時間の内訳を図に示す。GPU を用いることで CPU を用いた場合よりも全体で約 14.9 倍の高速化を実現した。また、CPU・GPU コードの収束解が一致し、実験値[1]とよく一致することを確認した。以上より、GPGPU を用いた決定論的手法に基づくα固有値計算の高速化を実現することができた。

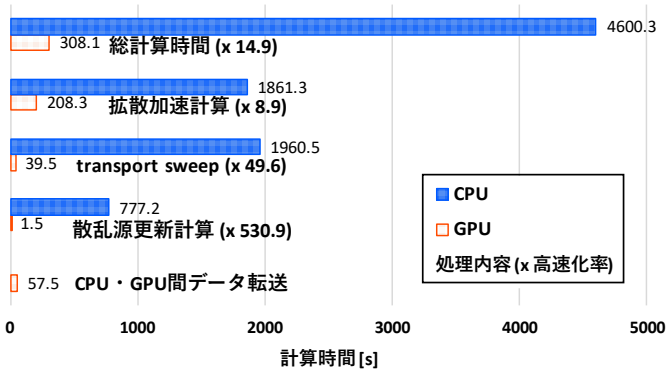


図 計算時間内訳の比較

参考文献 [1] K. Kobayashi et al., *J. Nucl. Sci. Technol.*, **3**(7), pp. 275–288 (1966).

口頭発表: 1. 山口響, 他, 日本原子力学会 2023 年春の年会, IK09, 3 月 13–15 日 (2023); **2.** 山口響, 他, 日本原子力学会 2023 年秋の大会, 1M10, 9 月 6–8 日 (2023); **3.** H. Yamaguchi, et al., *Proc. RPHA2023*, B-2-9, Gyeongju, Korea, Oct. 24–26 (2023); **4.** 山口響, 他, 日本原子力学会 2024 年春の年会, 3 月 26–28 日 (2024) (発表予定);

目次

第1章　序論

1.1　背景

軽水炉の詳細設計などに利用される軽水の中性子散乱断面積データには、広い入射中性子エネルギーと媒質温度範囲(解析対象が軽水炉の場合には、それぞれ1×10^{-5} eV~20 MeV、293.6~600 Kほど)にわたり信頼できるデータが必要とされる。特に、媒質の熱エネルギーと同程度のエネルギーをもつ中性子は熱中性子と呼ばれ、軽水炉における核分裂反応への寄与が大きいことから、熱中性子の散乱断面積は軽水炉解析において重要なデータとされる。

軽水素 ^{1}H の熱中性子の散乱においては、水の分子間振動などの水特有の複雑な分子運動が散乱後の中性子のエネルギー分布に影響する[1]ことから、それらを考慮する熱中性子散乱則(Thermal Scattering Law, TSL)データは軽水の熱中性子散乱断面積の評価値に大きく寄与する。そのため、実験結果を用いたデータ同化による TSL データの更新が数値解析精度の向上のために重要とされる[2]。しかし、核燃料を用いた臨界実験結果をデータ同化に利用する場合、^{235}U などの核燃料に含まれる核種が実効増倍率k_{eff}に強い影響を与えることから、軽水の TSL を選択的に改善することは困難であった[3],[4]。

そこで、未臨界体系でも測定可能な核特性である即発中性子減衰定数αの測定結果と数値解を、^{1}H の TSL データなどの評価済み核データの更新に活用することが提案されている[3],[5]。

即発中性子減衰定数αとは、体系内に存在する中性子数が$1/e$に減少するまでの時定数の逆数であり、パルス中性子法[6],[7]などにより測定することができる。パルス中性子法では、加速器中性子源などを用いて未臨界体系に周期的かつパルス状に中性子を打ち込むことを繰り返し、中性子計数率の時間変化を測定することでαを得る。パルス中性子法により測定された中性子計数率の時間変化の例を図 1.1 に示す。図 1.1 において即発中性子減衰定数αは指数関数的に減衰する中性子計数率の時定数の逆数に対応する。

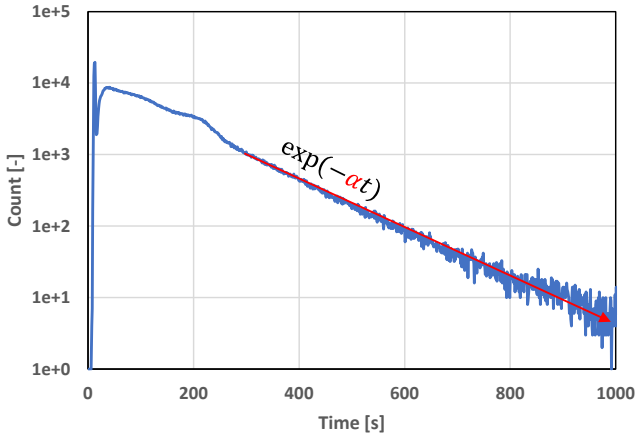


図 1.1 パルス中性子実験における中性子計数率の時間変化の例 [8]

上述したようにαは未臨界体系でも測定可能であることから、αを用いることで核燃料を含まない水槽体系のような極めて単純な体系を対象とした実験結果と数値解析結果を用いたデータ同化により、^{235}U などの核分裂性核種による影響を排除しつつ、水の ^{1}H TSL データが改善できる可能性があると期待されている[3]。また、核分裂性核種が存在する軽水減速体系においても、αはk_{eff}と強い相関を持つことから、k_{eff}不確かさやバイアスの低減も可能である[9],[10]と見込まれている。

ここで、水槽体系を対象とした即発中性子減衰定数αの数値計算について考える。^{1}H による中性子の散乱反応は強い非等方性を持つことから、水分子が大量に含まれる比較的小さな水槽体系を対象とした計算では、臨界体系を対象としたk_{eff}計算などと同様に非等方散乱中性子源の高精度な取扱が必要となる。これはS_N法[11]のような中性子飛行方向を考慮した中性子輸送計算手法により可能であり、水のみからなる体系を対象としたS_N法に基づくα固有値計算コードが先行研究[12]で試作されている。しかし、これらの決定論的手法に基づく計算手法では多数回の反復輸送計算が必要であり、大きな計算コストを伴う。ゆえに、少ない飛行方向分割数で高精度な計算が可能な角度分点セットや、拡散加速法などによる収束加速、反復計算自体の高速化が求められる。

以上で述べた課題を解決するため、本研究では計算手法の改良とともに、GPGPU(General-Purpose computing on Graphic Processing Units)と呼ばれる、GPU(Graphic Processing Unit)を計算の高速化などの画像処理以外の目的で利用する技術[13]に注目した。

1.2 本研究の目的

GPU は本来、画像処理を主に担う演算装置であるが、行列計算などのある特定の計算においては、数千のスレッドによる並列計算と広いメモリ帯域幅を活用して CPU を凌駕する性能を発揮することができる。このように GPU を画像処理以外の目的で利用する技術は GPGPU と呼ばれ、GPGPU は近年気候や流体、天体、分子動力学のシミュレーションに代表される科学技術計算や、機械学習など広い分野に活用されており、炉物理計算分野においても活用が進んでいる[14],[15],[16]。そこで本研究では、GPGPU を活用してS_N法に基づくα固有値計算を高速化することを目的とする。

より具体的には、S_N法の計算において大きな計算コストを要する非等方散乱中性子源更新及び transport sweep の GPGPU による高速化を図る。また、S_N法の反復計算の収束加速のため、拡散加速法を実装する。さらに、拡散加速法において大きな計算コストを要する、散乱中性子源更新と内部反復計算を GPGPU により高速化することも図る。

1.3　本論文の構成

本論文は 5 章構成である。

第 2 章では、S_N法に基づいたα固有値計算理論について理解を促すため、S_N法に基づくα固有値計算理論について述べる。まず、S_N法に基づくα固有値方程式の数値解法とS_N法の収束を加速するための拡散加速法[17]について述べる。また、S_N法及び拡散加速計算の数値不安定性の改善のために導入した delta-tracking 法[18]について説明する。

第 3 章では、GPGPU を用いたS_N法に基づくα固有値計算コードの開発について述べる。まず、GPU を科学技術計算などに応用する技術である GPGPU の概要を説明する。次に、GPGPU を用いた核計算コード開発において一般に課題となる事柄について述べる。そして、S_N法に基づくα固有値計算及びα固有値拡散加速計算を GPGPU により実装する手法について、それぞれ具体的に述べる。

第 4 章では、第 2 章及び第 3 章に基づき開発した、GPGPU を用いたS_N法に基づくα固有値計算コードの妥当性や高速化の効果を検証する。まず、本章の妥当性確認及び検証における計算条件について述べる。次に、先行研究によるα実験値と、開発した CPU 及び GPU コードによるαの計算結果を比較することで、開発したコードの妥当性確認及び検証を実施する。また、熱中性子散乱則に起因するα計算値の不確かさについても評価する。そして、開発した CPU 及び GPU コードの計算時間を比較し、GPU コードの有効性を検証する。

第 5 章では、本論文のまとめと今後の課題を述べる。

1.4　参考文献

[1] J. VAIBHAV, “Theoretical and Experimental Approach Towards Generation of Thermal Scattering Law for Light Water,” tel-02390769, Université de Lille (2018).

[2] D. ROCHMAN, A. VASILIEV, H. FERROUKHI, et al., “Impact of H in H_2O Thermal Scattering Data on Criticality Calculation: Uncertainty and Adjustment,” *EPJ Nuclear Sci. Technol.*, **8**, 3 (2022); https://doi.org/10.1051/epjn/2021028.

[3] Y. HARADA, H. YAMAGUCHI, T. ENDO, et al., “Uncertainty Quantification of Prompt Neutron Decay Constant α due to the Thermal Neutron Scattering Law of Water,” *Proc. M&C 2023*, Ontario, Canada, Aug. 13–17, 2023, American Nuclear Society (2023).

[4] Y. HARADA, H. YAMAGUCHI, T. ENDO, et al., “Data Assimilation Using Prompt Neutron Decay Constant for Water to Reduce Uncertainties due to Thermal Neutron Scattering Law,” *Proc. ICNC 2023*, Sendai, Japan, Oct. 1–6, 2023, ICNC2023 organizing committee (2023).

[5] T. ENDO, A. NOGUCHI, A. YAMAMOTO, et al., “Perturbation-Theory-Based Sensitivity Analysis of Prompt Neutron Decay Constant for Water-Only System”, *Transactions of American Nuclear Society*, 124, pp.184-187 (2021).

[6] B. E. SIMMONS and J. S. KING, “A Pulsed Neutron Technique for Reactivity Determination,” *Nuclear Science and Engineering*, **3**, 5, 595 (1958); https://doi.org/10.13182/NSE3-595-608.

[7] KOBAYASHI K, SEKI Y, MIZOO N, et al. Measurement and Calculation of Neutron Diffusion Parameters in Water. *Journal of Nuclear Science and Technology*, **3**, 275 (1966).

[8] F. NISIOKA. “炉物理実験手法に対する Dynamic Mode Decomposition の応用,” MS Thesis, Nagoya University, Department of Energy Engineering (May 2022).

[9] T. ENDO and A. YAMAMOTO, “Data Assimilation Using Subcritical Measurement of Prompt Neutron Decay Constant,” *Nuclear Science and Engineering*, **194**, 11, 1089 (2020); https://doi.org/10.1080/00295639.2020.1720499.

[10] Y. HARADA, H. YAMAGUCHI, T. ENDO, et al., “即発中性子減衰定数を用いたデータ同化による軽水の熱中性子散乱則に起因した不確かさの低減,” 日本原子力学会 2024 春の年会, Higashiosaka, Japan, Mar. 26–28, 2024.

[11] D. G. CACUCI, Ed., *Handbook of nuclear engineering*, Vol. 2, Springer, New York ; London (2010).

[12] A. NOGUCHI. “水体系における即発中性子減衰定数の決定論的数値解析手法に関する検討,” BS Thesis, Nagoya University, Department of Energy Science and Engineering (Feb. 2022).

[13] R. VUDUC and J. CHOI, “A Brief History and Introduction to GPGPU,” *Modern Accelerator Technologies for Geographic Information Science*, X. Shi, V. Kindratenko, and C. Yang, Eds., pp. 9–23, Springer US, Boston, MA (2013); https://doi.org/10.1007/978-1-4614-8745-6_2.

[14] Y. KODAMA, “GPU を用いた MOC の高速化に関する研究,” MS Thesis, Nagoya University, Materials, Physics and Energy Engineering (Feb. 2010).

[15] S. JEON et al., “Methods and Performance of a GPU-Based Pinwise Two-Step Nodal Code VANGARD,” *Progress in Nuclear Energy*, **156**, 104528 (2023); https://doi.org/10.1016/j.pnucene.2022.104528.

[16] T. OKUBO, T. ENDO, and A. YAMAMOTO, “An Efficient Execution of Monte Carlo Simulation Based on Delta-Tracking Method Using GPUs,” *Journal of Nuclear Science and Technology*, **54**, 1, 30 (2017); https://doi.org/10.1080/00223131.2016.1202793.

[17] A. ZHU, M. JARRETT, Y. XU, et al., “An optimally diffusive Coarse Mesh Finite Difference Method to Accelerate Neutron Transport Calculations,” *Annals of Nuclear Energy*, **95**, 116 (2016); https://doi.org/10.1016/j.anucene.2016.05.004.

[18] J. LEPPÄNEN, “Performance of Woodcock Delta-Tracking in Lattice Physics Applications Using the Serpent Monte Carlo Reactor Physics Burnup Calculation Code,” *Annals of Nuclear Energy*, **37**, 5, 715 (2010); https://doi.org/10.1016/j.anucene.2010.01.011.

第2章　S_N法に基づくα固有値計算理論

2.1　本章の概要

本研究では、中性子の飛行方向及び非等方散乱を考慮したα固有値計算のために、中性子輸送計算手法の 1 つであるS_N法(離散座標法)[1],[2]を利用する。S_N法に基づくα固有値計算理論について理解を促すため、本章ではS_N法に基づくα固有値計算理論について述べる。2.2 節では、S_N法に基づくα固有値方程式の数値解法について述べ、2.3 節ではS_N法の収束を加速するための拡散加速[3]について述べる。2.4 節では、S_N法及び拡散加速計算の数値不安定性の改善のために導入した delta-tracking 法[4]について説明する。

2.2　S_N法に基づくα固有値方程式の数値解法

本節では、S_N法に基づくα固有値方程式の数値解法について述べる。2.2.1 項では輸送理論に基づくα固有値方程式を示す。それに続いて、2.2.2 項で多群近似によるエネルギーの離散化を、2.2.3 項で飛行方向の離散化、2.2.4 項では空間離散化及び差分化について、それぞれ説明する。2.2.5 項では境界条件の取り扱いについて、2.2.6 項では非等方散乱成分の取り扱いについて述べ、2.2.7 項では角度分点セットについて述べる。2.2.8 項では導出したα固有値方程式の数値解を得る具体的な方法について述べ、2.2.9 項ではそれに必要な transport sweep と呼ばれる操作について述べる。

2.2.1　輸送理論に基づくα固有値方程式

中性子に関する時間依存のボルツマン輸送方程式は次式のように表される[1]

$$\frac{1}{v(E)}\frac{\partial}{\partial t}\psi(\boldsymbol{r},\boldsymbol{\Omega},E,t)+\boldsymbol{\Omega}\cdot\boldsymbol{\nabla}\psi(\boldsymbol{r},E,\boldsymbol{\Omega},t)+\Sigma_{\mathrm{t}}(\boldsymbol{r},E,t)\psi(\boldsymbol{r},E,\boldsymbol{\Omega},t)=Q(\boldsymbol{r},\boldsymbol{\Omega},E,t) \tag{2.1}$$

$\boldsymbol{r}$	: 位置
$\boldsymbol{\Omega}$	: 中性子飛行方向
E	: 中性子エネルギー
t	: 時間
$v(E)$	: 中性子速度
$\psi(\boldsymbol{r},E,\boldsymbol{\Omega},t)$	: 角度中性子束
$\Sigma_t(\boldsymbol{r},E,t)$	: 巨視的全断面積
$Q(\boldsymbol{r},\boldsymbol{\Omega},E,t)$	: 中性子源

ここで、即発中性子減衰定数αが指数関数的に減少すると仮定すると、角度中性子束ψの時間微分は以下のように表現できる。

$$\psi\propto\exp(-\alpha t)\Rightarrow\frac{\partial\psi}{\partial t}=-\alpha\psi \tag{2.2}$$

これを式(2.1)に代入すると、輸送理論に基づくα固有値方程式が次のように導出できる[5]。

$$\boldsymbol{\Omega}\cdot\boldsymbol{\nabla}\psi(\boldsymbol{r},E,\boldsymbol{\Omega})+\left\{\Sigma_{\mathrm{t}}(\boldsymbol{r},E)-\frac{\alpha}{v(E)}\right\}\psi(\boldsymbol{r},E,\boldsymbol{\Omega})=Q(\boldsymbol{r},\boldsymbol{\Omega},E) \tag{2.3}$$

2.2.2 多群近似によるエネルギーの離散化

エネルギーの離散化では、中性子エネルギーをいくつかのグループに分ける多群近似が用いられる。エネルギーg群の角度中性子束、中性子源、巨視的全断面積、中性子速度を式(2.4)–(2.7)のように定義する[1]。

$$\psi_g(\boldsymbol{r},\boldsymbol{\Omega})=\int_{E_g}^{E_{g-1}}\psi(\boldsymbol{r},E,\boldsymbol{\Omega})dE \tag{2.4}$$

$$Q_g(\boldsymbol{r},\boldsymbol{\Omega})=\int_{E_g}^{E_{g-1}}Q(\boldsymbol{r},E,\boldsymbol{\Omega})dE \tag{2.5}$$

$$\Sigma_{\mathrm{t},g}(\boldsymbol{r})=\frac{\int_{E_g}^{E_{g-1}}\Sigma_{\mathrm{t}}(\boldsymbol{r},E)\phi(\boldsymbol{r},E)dE}{\int_{E_g}^{E_{g-1}}\phi(\boldsymbol{r},E)dE} \tag{2.6}$$

$$\frac{1}{v_g(\boldsymbol{r})}=\frac{\int_{E_g}^{E_{g-1}}\phi(\boldsymbol{r},E)/v(E)dE}{\int_{E_g}^{E_{g-1}}\phi(\boldsymbol{r},E)dE} \tag{2.7}$$

ここで、$\phi(\boldsymbol{r},E)$は全中性子束であり、次式により定義される。

$$\phi(\boldsymbol{r},E)=\int_0^{4\pi}\psi(\boldsymbol{r},E,\boldsymbol{\Omega})d\boldsymbol{\Omega} \tag{2.8}$$

多群近似により、ボルツマン輸送方程式は次のようにNG個の連立方程式の形に書き直すことができる。

$$\boldsymbol{\Omega}\cdot\boldsymbol{\nabla}\psi_g(\boldsymbol{r},\boldsymbol{\Omega})+\left\{\Sigma_{\mathrm{t},g}(\boldsymbol{r})-\frac{\alpha}{v_g}\right\}\psi_g(\boldsymbol{r},\boldsymbol{\Omega})=Q_g(\boldsymbol{r},\boldsymbol{\Omega})\ \ ,1\le g\le NG \tag{2.9}$$

2.2.3 飛行方向の離散化

S_N法では、全立体角4πをND個の飛行方向で代表させることで式(2.9)を飛行方向について離散化する[1]。これにより、次式が導出できる。

$$\boldsymbol{\Omega}_d\cdot\boldsymbol{\nabla}\psi_{g,d}(\boldsymbol{r})+\left\{\Sigma_{\mathrm{t},g}(\boldsymbol{r})-\frac{\alpha}{v_g}\right\}\psi_{g,d}(\boldsymbol{r})=Q_{g,d}(\boldsymbol{r})\ \ ,1\le d\le ND \tag{2.10}$$

このとき、全中性子束は、飛行方向に対して解析的に積分する代わりに、各飛行方向$\boldsymbol{\Omega}_d$に対する重みをw_dとしたガウス求積により求められる[1],[2]。

$$\phi_g(\boldsymbol{r})=4\pi\sum_{d=1}^{ND}w_d\psi_{g,d}(\boldsymbol{r}) \tag{2.11}$$

上式において、w_dの総和は 1 となるように規格化されている。

$$\sum_{d=1}^{ND} w_d = 1 \tag{2.12}$$

ND個のw_dと$\boldsymbol{\Omega}_d$の組は角度分点セットと呼ばれる。角度分点セットについては、2.2.7 項で詳しく述べる。

2.2.4 空間離散化と差分化

本節では 3 次元直交座標系を仮定して空間の離散化を行う。3 次元直交座標系における$\boldsymbol{r}, \boldsymbol{\Omega}_d$は以下のように表される。

$$\boldsymbol{r} = x\boldsymbol{e}_x + y\boldsymbol{e}_y + z\boldsymbol{e}_z \tag{2.13}$$

$\boldsymbol{e_x}$ ：x軸方向の単位ベクトル
$\boldsymbol{e_y}$ ：y軸方向の単位ベクトル
$\boldsymbol{e_z}$ ：z軸方向の単位ベクトル

$$\boldsymbol{\Omega}_d = \mu_d \boldsymbol{e}_x + \eta_d \boldsymbol{e}_y + \xi_d \boldsymbol{e}_z \tag{2.14}$$

μ ：x軸方向の方向余弦
η ：y軸方向の方向余弦
ξ ：z軸方向の方向余弦

式(2.13), (2.14)より、式(2.10)は次式のように書き直せる。

$$\left(\mu_d \frac{\partial}{\partial x} + \eta_d \frac{\partial}{\partial y} + \xi_d \frac{\partial}{\partial z}\right)\psi_{g,d}(\boldsymbol{r}) + \left\{\Sigma_{\mathrm{t},g}(\boldsymbol{r}) - \frac{\alpha}{v_g}\right\}\psi_{g,d}(\boldsymbol{r}) = Q_{g,d}(\boldsymbol{r}) \tag{2.15}$$

ここで、計算体系をx, y, z方向にそれぞれNX, NY, NZ個のメッシュに分割するとき、各i, j, k番目のメッシュについて、

- メッシュの中点の座標: (x_i, y_j, z_k)
- メッシュのx, y, z方向長さ: $\Delta x_i, \Delta y_j, \Delta z_k$
- メッシュ内で多群断面積は一定
- メッシュ内の平均角度中性子束$\psi_{g,d,i,j,k}$:

$$\psi_{g,d,i,j,k} = \frac{1}{\Delta x_i \Delta y_j \Delta z_k}\int_{x_i^-}^{x_i^+}\int_{y_j^-}^{y_j^+}\int_{z_k^-}^{z_k^+}\psi_{g,d}(\boldsymbol{r})\,dxdydz \tag{2.16}$$

- メッシュ内の平均角度中性子源$Q_{g,d,i,j,k}$:

$$Q_{g,d,i,j,k} = \frac{1}{\Delta x_i \Delta y_j \Delta z_k}\int_{x_i^-}^{x_i^+}\int_{y_j^-}^{y_j^+}\int_{z_k^-}^{z_k^+}Q_{g,d}(\boldsymbol{r})\,dxdydz \tag{2.17}$$

とする。ただし、上添え字の$\pm$は各メッシュ各軸の正、負方向の境界を表す。式(2.15)を各メッシュについて積分することで、差分化された次式を得る[1]。

$$\frac{\mu_d}{\Delta x_i}\left(\psi^{x+}_{g,d,i,j,k}-\psi^{x-}_{g,d,i,j,k}\right)+\frac{\eta_d}{\Delta y_j}\left(\psi^{y+}_{g,d,i,j,k}-\psi^{y-}_{g,d,i,j,k}\right)+\frac{\xi_d}{\Delta z_k}\left(\psi^{k+}_{g,d,i,j,k}-\psi^{k-}_{g,d,i,j,k}\right)$$
$$+\left\{\Sigma_{t,g,i,j,k}-\frac{\alpha}{v_g}\right\}\psi_{g,d,i,j,k}=Q_{g,d,i,j,k} \tag{2.18}$$

$\psi^{x\pm}_{g,d,i,j,k}$ ：x方向±側境界面における角度中性子束

$\psi^{y\pm}_{g,d,i,j,k}$ ：z方向±側境界面における角度中性子束

$\psi^{z\pm}_{g,d,i,j,k}$ ：z方向±側境界面における角度中性子束

ここまでで、エネルギー、飛行方向、空間について離散化されたα固有値方程式(2.18)が得られた。しかし、例えば流入角度中性子束として$\psi^{x-}_{g,d,i,j,k},\psi^{y-}_{g,d,i,j,k},\psi^{z-}_{g,d,i,j,k}$が既知であるとしても、$\psi^{x+}_{g,d,i,j,k},\psi^{y+}_{g,d,i,j,k},\psi^{z+}_{g,d,i,j,k},\psi_{g,d,i,j,k}$は未知であり、未知数の数に対して式の数が不足しており、方程式を解くためには別の関係式が必要とされる。本研究ではこの関係式としてダイヤモンド差分法[1],[2],[6]を利用する。ダイヤモンド差分法では式(2.18)に対して次式に示す近似を適用する。

$$\begin{aligned}\psi_{g,d,i,j,k}&=\frac{\psi^{x-}_{g,d,i,j,k}+\psi^{x-}_{g,d,i,j,k}}{2}\\&=\frac{\psi^{y-}_{g,d,i,j,k}+\psi^{y+}_{g,d,i,j,k}}{2}\\&=\frac{\psi^{z-}_{g,d,i,j,k}+\psi^{z+}_{g,d,i,j,k}}{2}\end{aligned} \tag{2.19}$$

ここで、各メッシュに対するx,y,z方向からの流入、流出角度中性子束をそれぞれ次のように定義する(図 2.1)。

$$\psi^{xin}_{g,d,i,j,k}=\begin{cases}\psi^{x-}_{g,d,i,j,k} & \text{if } \mu_d>0\\ \psi^{x+}_{g,d,i,j,k} & \text{if } \mu_d<0\end{cases},\quad \psi^{xout}_{g,d,i,j,k}=\begin{cases}\psi^{x+}_{g,d,i,j,k} & \text{if } \mu_d>0\\ \psi^{x-}_{g,d,i,j,k} & \text{if } \mu_d<0\end{cases}$$
$$\psi^{yin}_{g,d,i,j,k}=\begin{cases}\psi^{y-}_{g,d,i,j,k} & \text{if } \eta_d>0\\ \psi^{y+}_{g,d,i,j,k} & \text{if } \eta_d<0\end{cases},\quad \psi^{yout}_{g,d,i,j,k}=\begin{cases}\psi^{y+}_{g,d,i,j,k} & \text{if } \eta_d>0\\ \psi^{y-}_{g,d,i,j,k} & \text{if } \eta_d<0\end{cases} \tag{2.20}$$
$$\psi^{zin}_{g,d,i,j,k}=\begin{cases}\psi^{z-}_{g,d,i,j,k} & \text{if } \xi_d>0\\ \psi^{z+}_{g,d,i,j,k} & \text{if } \xi_d<0\end{cases},\quad \psi^{zout}_{g,d,i,j,k}=\begin{cases}\psi^{z+}_{g,d,i,j,k} & \text{if } \xi_d>0\\ \psi^{z-}_{g,d,i,j,k} & \text{if } \xi_d<0\end{cases}$$

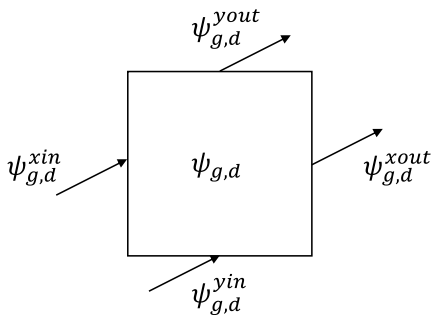


図 2.1　2 次元平面での平均角度中性子束と流入・流出角度中性子束

式(2.18), (2.19), (2.20)より、各メッシュの平均角度中性子束を次式により表すことができる。

$$\psi_{g,d,i,j,k} = \frac{Q_{g,d,i,j,k} + 2\left(\frac{|\mu_d|}{\Delta x_{\mathrm{i}}}\psi^{xin}_{g,d,i,j,k} + \frac{|\eta_d|}{\Delta y_j}\psi^{yin}_{g,d,i,j,k} + \frac{|\xi_d|}{\Delta z_k}\psi^{zin}_{g,d,i,j,k}\right)}{\left(\Sigma_{\mathrm{t},g,i,j,k} - \frac{\alpha}{v_g}\right) + 2\left(\frac{|\mu_d|}{\Delta x_{\mathrm{i}}} + \frac{|\eta_d|}{\Delta y_j} + \frac{|\xi_d|}{\Delta z_k}\right)} \tag{2.21}$$

$Q_{g,d,i,j,k}, \psi^{xin}_{g,d,i,j,k}, \psi^{yin}_{g,d,i,j,k}, \psi^{zin}_{g,d,i,j,k}$が既知であれば、式(2.21)により平均角度中性子束$\psi_{g,d,i,j,k}$を計算することができる。そして、$\psi_{g,d,i,j,k}$を計算することができれば、式(2.19), (2.20)より流出角度中性子束を次式により計算することができる。

$$\begin{aligned}
\psi^{xout}_{g,d,i,j,k} &= 2\psi_{g,d,i,j,k} - \psi^{xin}_{g,d,i,j,k} \\
\psi^{yout}_{g,d,i,j,k} &= 2\psi_{g,d,i,j,k} - \psi^{yin}_{g,d,i,j,k} \\
\psi^{zout}_{g,d,i,j,k} &= 2\psi_{g,d,i,j,k} - \psi^{zin}_{g,d,i,j,k}
\end{aligned} \tag{2.22}$$

2.2.5 境界条件

本研究では、簡単のため、真空境界条件のみを取り扱う。真空境界条件は、体系境界面からの入射角度中性子束が 0 であることに相当し、以下のように与えられる。

$$\begin{cases} \psi^{xin}_{g,d,0,j,k} = 0 & \text{if } \mu_d > 0 \\ \psi^{xin}_{g,d,NX,j,k} = 0 & \text{if } \mu_d < 0 \end{cases} \text{(Vacuum B. C.)} \tag{2.23}$$

$$\begin{cases} \psi^{yin}_{g,d,i,0,k} = 0 & \text{if } \eta_d > 0 \\ \psi^{yin}_{g,d,i,NY,k} = 0 & \text{if } \eta_d < 0 \end{cases} \text{(Vacuum B. C.)} \tag{2.24}$$

$$\begin{cases} \psi^{zin}_{g,d,i,j,0} = 0 & \text{if } \xi_d > 0 \\ \psi^{zin}_{g,d,i,j,NZ} = 0 & \text{if } \xi_d < 0 \end{cases} \text{(Vacuum B. C.)} \tag{2.25}$$

2.2.6　非等方散乱成分の取扱

散乱反応には非等方性があり、例えば ^{1}H のような軽い核種との散乱反応では、中性子は前方に偏って散乱される。本項では、その散乱反応の非等方性の取り扱いについて述べる。

まず、実球面調和関数$R_l^m(\boldsymbol{\Omega})$は球面調和関数$Y_l^m(\boldsymbol{\Omega})$を用いて次のように表現される[2]。ただし、$Y_l^m(\boldsymbol{\Omega})^*$は球面調和関数$Y_l^m(\boldsymbol{\Omega})$の複素共役を表す。

$$R_l^m(\boldsymbol{\Omega}) \equiv \begin{cases} (Y_l^m(\boldsymbol{\Omega}) - Y_l^m(\boldsymbol{\Omega})^*)/\sqrt{2} & \text{if } m < 0 \\ Y_l^m(\Omega) & \text{if } m = 0 \\ (Y_l^m(\boldsymbol{\Omega}) + Y_l^{-m}(\boldsymbol{\Omega})^*)/\sqrt{2} & \text{if } m > 0 \end{cases} \tag{2.26}$$

実球面調和関数$R_l^m(\boldsymbol{\Omega})$は球面調和関数$Y_l^m(\boldsymbol{\Omega})$と同様に次の直交規格化の条件式を満たす(次式におけるδはクロネッカーのデルタである)。

$$\int_0^{4\pi} R_l^m(\boldsymbol{\Omega}) R_{l'}^{m'}(\boldsymbol{\Omega}) d\boldsymbol{\Omega} = \frac{4\pi}{2l+1} \delta_{ll'} \delta_{mm'} \tag{2.27}$$

そして、角度中性子束及び中性子源を、次のように実球面調和関数$R_l^m(\boldsymbol{\Omega})$で展開することを考える。

$$\psi(\boldsymbol{r}, \boldsymbol{\Omega}, E) = \sum_{l=0}^{NL} \frac{2l+1}{4\pi} \sum_{m=-l}^{l} \phi_l^m(\boldsymbol{r}, E) R_l^m(\boldsymbol{\Omega}) \tag{2.28}$$

$$Q(\boldsymbol{r}, \boldsymbol{\Omega}, E) = \sum_{l=0}^{NL} \frac{2l+1}{4\pi} \sum_{m=-l}^{l} Q_l^m(\boldsymbol{r}, E) R_l^m(\boldsymbol{\Omega}) \tag{2.29}$$

$\phi_l^m(\boldsymbol{r}, \boldsymbol{\Omega}, E)$　：各展開次数に対応する中性子束モーメント(展開係数)

$Q_l^m(\boldsymbol{r}, \boldsymbol{\Omega}, E)$　：各展開次数に対応する中性子源モーメント(展開係数)

このとき、中性子束モーメントは次式のように表せる。

$$\phi_l^m(\boldsymbol{r}, E) = \int_0^{4\pi} \psi(\boldsymbol{r}, \boldsymbol{\Omega}, E) R_l^m(\boldsymbol{\Omega}) d\boldsymbol{\Omega} \tag{2.30}$$

また、式(2.28)を式(2.8)に代入し、式(2.27)の直交関係を用いると、次式を得る。

$$\phi(\boldsymbol{r},E)=\phi_0^0(\boldsymbol{r},E) \tag{2.31}$$

すなわち、式(2.28)の展開の第 1 項は全中性子束に等しい。式(2.28)〜(2.31)をエネルギー、飛行方向、空間について離散化した形にすると、以下のように表せる。

$$\psi_{g,d,i,j,k}=\sum_{l=0}^{NL}\frac{2l+1}{4\pi}\sum_{m=-l}^{l}\phi_{l,g,i,j,k}^{m}R_l^m(\mu_d,\eta_d,\xi_d) \tag{2.32}$$

$$Q_{g,d,i,j,k}=\sum_{l=0}^{NL}\frac{2l+1}{4\pi}\sum_{m=-l}^{l}Q_{l,g,i,j,k}^{m}R_l^m(\mu_d,\eta_d,\xi_d) \tag{2.33}$$

$$\phi_{l,g,i,j,k}^{m}=\sum_{d=1}^{ND}w_d\psi_{g,d,i,j,k}R_l^m(\mu_d,\eta_d,\xi_d) \tag{2.34}$$

$$\phi_{g,i,j,k}=\phi_{0,g,i,j,k}^{0} \tag{2.35}$$

また、非等方散乱断面積$\Sigma_{\mathrm{s}}(\boldsymbol{r},\boldsymbol{\Omega}'\to\boldsymbol{\Omega},E'\to E)$は、次のように実球面調和関数$R_l^m(\boldsymbol{\Omega})$で展開することができる[2]。

$$\Sigma_{\mathrm{s}}(\boldsymbol{r},\boldsymbol{\Omega}'\to\boldsymbol{\Omega},E'\to E)=\sum_{l=0}^{NL}\frac{2l+1}{4\pi}\sum_{m=-l}^{l}\Sigma_{sl}(\boldsymbol{r},E'\to E)R_l^m(\boldsymbol{\Omega}')R_l^m(\boldsymbol{\Omega}) \tag{2.36}$$

$$\begin{aligned}\Sigma_{sl}(\boldsymbol{r},E'\to E)&=\int_0^{\pi}P_l(\cos\theta)\Sigma_{\mathrm{s}}(\boldsymbol{r},\cos\theta,E'\to E)\sin\theta\,d\theta\\&=\int_{-1}^{1}P_l(\mu)\Sigma_{\mathrm{s}}(\boldsymbol{r},\mu,E'\to E)\,d\mu\end{aligned} \tag{2.37}$$

μ　：散乱角度の方向余弦 $\mu=\cos\theta=\boldsymbol{\Omega}\cdot\boldsymbol{\Omega}'$

$P_l(\mu)$　：ルジャンドル多項式

$\Sigma_{sl}(\boldsymbol{r},E'\to E)$は巨視的散乱断面積をルジャンドル多項式で展開したときのl次の展開係数に相当する。例えば、$l=0,1$のとき以下のように表される。

$$\Sigma_{\mathrm{s}0}(\boldsymbol{r},E'\to E)=\int_{-1}^{1}P_0(\mu)\Sigma_{\mathrm{s}}(\boldsymbol{r},\mu,E'\to E)\,d\mu=\int_{-1}^{1}\Sigma_{\mathrm{s}}(\boldsymbol{r},\mu,E'\to E)\,d\mu \tag{2.38}$$

$$\begin{aligned}\Sigma_{\mathrm{s}1}(\boldsymbol{r},E'\to E)&=\int_{-1}^{1}P_1(\mu)\Sigma_{\mathrm{s}}(\boldsymbol{r},\mu,E'\to E)\,d\mu=\int_{-1}^{1}\mu\Sigma_{\mathrm{s}}(\mu)\,d\mu\\&=\frac{\int_{-1}^{1}\mu\Sigma_{\mathrm{s}}(\boldsymbol{r},\mu,E'\to E)\,d\mu}{\int_{-1}^{1}\Sigma_{\mathrm{s}}(\boldsymbol{r},\mu,E'\to E)\,d\mu}\int_{-1}^{1}\Sigma_{\mathrm{s}}(\boldsymbol{r},\mu,E'\to E)\,d\mu=\bar{\mu}\Sigma_{\mathrm{s}0}\end{aligned} \tag{2.39}$$

$\bar{\mu}$は散乱角度の平均方向余弦であり、次式により表される。

$$\bar{\mu}=\frac{\int_{-1}^{1}\mu\Sigma_{\mathrm{s}}(\boldsymbol{r},\mu,E'\to E)\,d\mu}{\int_{-1}^{1}\Sigma_{\mathrm{s}}(\boldsymbol{r},\mu,E'\to E)\,d\mu} \tag{2.40}$$

なお、本研究では非増倍体系すなわち核分裂のない体系を計算対象とするため、核分裂源は考慮しない。従って、中性子源$Q(\boldsymbol{r},\boldsymbol{\Omega},E)$は次式により表される。

$$Q(\boldsymbol{r},\boldsymbol{\Omega},E)=\int_0^{\infty}\int_0^{4\pi}\Sigma_{\mathrm{s}}(\boldsymbol{r},\boldsymbol{\Omega}'\rightarrow\boldsymbol{\Omega},E'\rightarrow E)\psi(\boldsymbol{r},\boldsymbol{\Omega},E)d\boldsymbol{\Omega}'dE' \tag{2.41}$$

式(2.36)を式(2.41)に代入して、次式を得る。

$$Q(\boldsymbol{r},\boldsymbol{\Omega},E)= \int_0^{\infty}\sum_{l=0}^{NL}\frac{2l+1}{4\pi}\sum_{m=-l}^{l}\Sigma_{sl}(\boldsymbol{r},E'\rightarrow E)R_l^m(\boldsymbol{\Omega})\left(\int_0^{4\pi}\psi(\boldsymbol{r},\boldsymbol{\Omega}',E')R_l^m(\boldsymbol{\Omega}')d\boldsymbol{\Omega}'\right)dE' \tag{2.42}$$

さらに、式(2.30)より上式は以下のように変形できる。

$$Q(\boldsymbol{r},\boldsymbol{\Omega},E)=\int_0^{\infty}\sum_{l=0}^{NL}\frac{2l+1}{4\pi}\sum_{m=-l}^{l}\Sigma_{sl}(\boldsymbol{r},E'\rightarrow E)\phi_l^m(\boldsymbol{r},E)R_l^m(\boldsymbol{\Omega})\,dE' \tag{2.43}$$

式(2.29)、(2.43)より、$Q_l^m(\boldsymbol{r},\boldsymbol{\Omega},E)$は次のように表現できる。

$$Q_l^m(\boldsymbol{r},\boldsymbol{\Omega},E)=\int_0^{\infty}\Sigma_{\mathrm{s}l}(\boldsymbol{r},E'\rightarrow E)\phi_l^m(\boldsymbol{r},E')\,dE' \tag{2.44}$$

上式をエネルギー、飛行方向、空間について離散化することで、次式を得る。

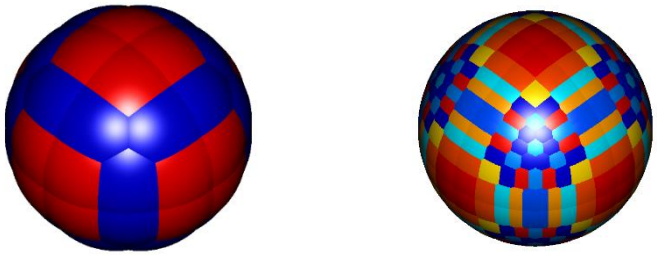
$$Q_{l,g,i,j,k}^m=\sum_{g'=1}^{NG}\Sigma_{\mathrm{s}l,g'\rightarrow g,i,j,k}\phi_{l,g',i,j,k}^m \tag{2.45}$$

$\Sigma_{sl,g'\rightarrow g}$　：g'群からg群へのl次の巨視的非等方散乱断面積

2.2.7　角度分点セット

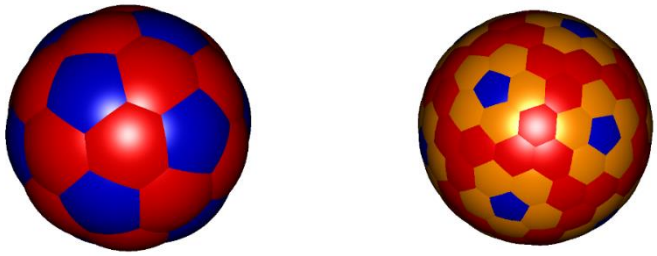
S_N法では、前項で示したような球面調和関数を用いた数値積分を正確に計算することが重要となる。ゆえに角度分点セットは、少ない中性子飛行方向分割数NDで、高精度に高次のl,mまで数値積分できることが望ましい。代表的な角度分点セットとしては、level symmetric 分点[1],[2](図 2.2)や Chebyshev-Legendre 分点[6],[7]などが挙げられる。

本研究では、少ない飛行方向分割数で高精度な球面調和関数の積分計算が可能である特徴を持つ icosahedral 分点[8](図 2.3)を利用する。level symmetric 分点では、20 次(飛行方向分割数 440, 図 2.2(ii))を超えると負の重みが出現するという問題があった[7]。そこで、方位角を等間隔に離散化し、極角を Gauss-Legendre 公式で離散化するような分点(例: Chebyshev-Legendre 分点)が開発されたが、このような分点では極付近に分点が集中してしまう問題が生じる。一方で Ahrens と Beylkin により提案された icosahedral 分点は、正二十面体の回転群を用いることで球面に対するほぼ最適な求積が可能な分点セットとなっており、分点は球面上でほぼ一様に分布する。先行研究[9]において、icosahedral 分点を利用することで、中性子輸送計算の射線効果が低減できると報告されている。



(i) 48 方向　　(ii) 440 方向

図 2.2 level symmetric 分点[10]



(i) 32 方向　　(ii) 132 方向

図 2.3 icosahedral 分点[10]

一方で、S_N法で icosahedral 分点を用いる場合には注意すべき点がある。通常の icosahedral 分点には、方向余弦μ,η,ξのいずれかが 0 となることがあり、これは 2.2.4 項や 2.2.5 項で述べた角度中性子束の取り扱いと整合性が取れなくなる問題がある。そこで、icosahedral 分点セットを極角と方位角方向にいくらか回転させ、方向余弦が 0 とならないようにした上で計算を実施する必要がある。また、回転後の icosahedral 分点は各象限について対称ではないため、完全反射境界条件を適用することが困難である点にも注意する必要がある。

2.2.8　反復解法

本項では、数値計算により即発中性子減衰定数αを得る方法について述べる。

即発中性子減衰定数αは式(2.18)で示したα固有値方程式の最小固有値に対応するが、式(2.18)を行列形式で表してそれに対し逆べき乗などを直接適用することは、係数行列が巨大で複雑な疎行列となることから、現実的ではない。よって、通常このような問題に対しては反復法が用いられる。反復法では以下に示すような逆べき乗法に相当する手順により最小のα固有値を得ることができる[5]。

1. 全中性子束とαの初期値$\phi_0^{0,(0)}$と$\alpha^{(0)}$を与える。$(l,m)=(0,0)$以外の高次の中性子束モーメント$\phi_l^{m,(0)}$はゼロとする。ここで、$\phi_0^{0,(0)}$は次式を満たすよう規格化する。

$$\alpha^{(n)}=\frac{1}{\sum_{g=1}^{NG}\sum_{k=1}^{NZ}\sum_{j=1}^{NY}\sum_{i=1}^{NX}\frac{\phi_{0\,g,i,j,k}^{0(n)}}{v_g}\Delta x_i\Delta y_j\Delta z_k} \tag{2.46}$$

2. n回目の外部反復を開始。
3. $g=1$とする。
4. $\phi_{l,g,i,j,k}^{m,(n-1)}$を用いて、式(2.45)及び(2.33)により$g$群の各メッシュに対し$Q_{l,g,i,j,k}^{m,(n)}$及び$Q_{g,d,i,j,k}^{(n)}$を計算する。
5. $Q_{g,d,i,j,k}^{(n)}$を用いて、transport sweep(次項で説明)により$\psi_{g,d,i,j,k}^{(n)}$を計算する。
   また、式(2.34)と$\psi_{g,d,i,j,k}^{(n)}$を用いて$\phi_{l,g,i,j,k}^{m,(n)}$を計算する。
6. 手順 4.と 5.を$g=NG$まで繰り返す。
7. 式(2.46)により$\alpha^{(n)}$を計算する。
8. $\alpha^{(n)}$と$\alpha^{(n-1)}$の相対差異が収束判定基準ε未満になるまで、手順 2.から 7.を繰り返す。

2.2.9 transport sweep

transport sweep とは、与えられた中性子源$Q_{g,d,i,j,k}$に対して、式(2.21)及び式(2.22)に基づき各飛行方向について角度中性子束$\psi_{g,d,i,j,k}$を更新する計算のことをいう。式(2.21)、(2.22)からわかるように、ある空間メッシュ(i,j,k)の流出角度中性子束$\psi^{xout}_{g,d,i,j,k}, \psi^{yout}_{g,d,i,j,k}, \psi^{zout}_{g,d,i,j,k}$を求めるには、その空間メッシュの平均角度中性子束$\psi_{g,d,i,j,k}$が必要で、平均角度中性子束を求めるには流入角度中性子束$\psi^{xin}_{g,d,i,j,k}, \psi^{yin}_{g,d,i,j,k}, \psi^{zin}_{g,d,i,j,k}$が必要である。ゆえに、transport sweep では以下の手順でそれぞれの値を評価する。

1. 流入角度中性子束：
   境界条件(式(**2.23**)–(**2.25**))により与えるか、
   流入側に隣接する空間メッシュからの流出角度中性子束を流入角度中性子束とする。
2. 平均角度中性子束：式(2.21)に基づき計算する。
3. 流出角度中性子束：式(2.22)に基づき計算する。
4. 流出側に隣接する空間メッシュへ移動し、最後のメッシュに到達するまで 1.から手順を繰り返す。

空間メッシュのどの面が流入、流出側となるかは、各方向余弦μ_d, η_d, ξ_dの正負の組み合わせによるため、空間メッシュを移動する順序は正負の組み合わせごとに変える必要がある。例えば$\mu_d > 0, \eta_d > 0, \xi_d > 0$の場合、$(i,j,k)$の計算は$(i-1,j,k), (i,j-1,k), (i,j,k-1)$の計算結果に依存するため、これを満たすようにメッシュを移動しながら計算する必要がある。例として、$\mu_d > 0, \eta_d > 0, \xi_d > 0$の場合の計算順序の一例を図に示す。

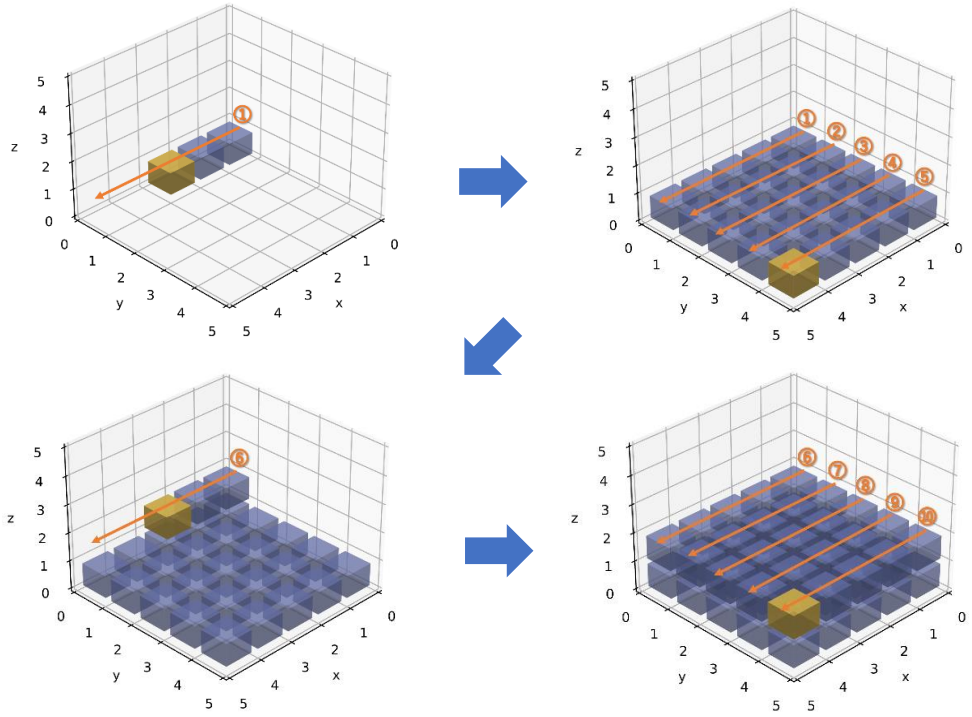


図 2.4 transport sweep の計算順序の例($\mu_d > 0, \eta_d > 0, \xi_d > 0$)

2.3 拡散加速法

S_N法のような中性子輸送計算手法は、自群散乱を陽に取り扱うため収束性が悪いことが知られている[1]。ゆえに実用的な中性子輸送計算では通常、収束性を改善するため様々な加速法を適用する必要がある。本節では、開発したコードに適用した加速法の 1 つである拡散加速法[3]について述べる。

2.3.1 項では、拡散加速法の基本となる拡散理論に基づくα固有値方程式を示す。2.3.2 項では拡散加速計算において、詳細計算(本研究の場合S_N法計算)で得られる正味の中性子流を再現するための中性子流補正係数について述べ、2.3.3 項では拡散加速計算で解くべきα固有値方程式と計算手順について述べる。

2.3.1 拡散理論に基づくα固有値方程式

拡散加速法は、詳細計算における正味の中性子流と反応率を保存するようにして拡散理論に基づいた方程式を解いた結果を利用することで詳細計算の収束性を高める手法である[11]。拡散加速法の基本となる拡散理論に基づくα固有値方程式をエネルギー及び空間について離散化、差分化したものを以下に示す[5],[11]。ただし、本研究では非増倍体系を計算対象とするため、核分裂源項を含んでいないことに注意されたい。

$$\frac{J^{x+}_{g,i,j,k}-J^{x-}_{g,i,j,k}}{\Delta x_i}+\frac{J^{y+}_{g,i,j,k}-J^{y-}_{g,i,j,k}}{\Delta y_j}+\frac{J^{z+}_{g,i,j,k}-J^{z-}_{g,i,j,k}}{\Delta z_k}+\Sigma_{\mathrm{t},g,i,j,k}\phi_{g,i,j,k}$$
$$=\frac{\alpha}{v_g}\phi_{g,i,j,k}+\sum_{g'=1}^{NG}\Sigma_{\mathrm{s},g'\to g,i,j,k}\phi_{g,i,j,k} \tag{2.47}$$

$$J^{x-}_{g,i,j,k}\approx-\frac{2D_{g,i-1,j,k}D_{g,i,j,k}}{D_{g,i-1,j,k}\Delta x_i+D_{g,i,j,k}\Delta x_{i-1}}\left(\phi_{g,i,j,k}-\phi_{i-1,j,k,g}\right) \tag{2.48}$$

$$J^{x+}_{g,i,j,k}\approx-\frac{2D_{g,i,j,k}D_{g,i+1,j,k}}{D_{g,i,j,k}\Delta x_{i+1}+D_{g,i+1,j,k}\Delta x_i}\left(\phi_{i+1,j,k,g}-\phi_{g,i,j,k}\right) \tag{2.49}$$

$$J^{y-}_{g,i,j,k}\approx-\frac{2D_{g,i,j-1,k}D_{g,i,j,k}}{D_{g,i,j-1,k}\Delta y_j+D_{g,i,j,k}\Delta y_{j-1}}\left(\phi_{g,i,j,k}-\phi_{i,j-1,k,g}\right) \tag{2.50}$$

$$J^{y+}_{g,i,j,k}\approx-\frac{2D_{g,i,j,k}D_{g,i,j+1,k}}{D_{g,i,j,k}\Delta y_{j+1}+D_{g,i,j+1,k}\Delta y_j}\left(\phi_{i,j+1,k,g}-\phi_{g,i,j,k}\right) \tag{2.51}$$

$$J^{z-}_{g,i,j,k}\approx-\frac{2D_{g,i,j,k-1}D_{g,i,j,k}}{D_{g,i,j,k-1}\Delta z_k+D_{g,i,j,k}\Delta z_{k-1}}\left(\phi_{g,i,j,k}-\phi_{i,j,k-1,g}\right) \tag{2.52}$$

$$J^{z+}_{g,i,j,k}\approx-\frac{2D_{g,i,j,k}D_{g,i,j,k+1}}{D_{g,i,j,k}\Delta z_{k+1}+D_{g,i,j,k+1}\Delta z_k}\left(\phi_{i,j,k+1,g}-\phi_{g,i,j,k}\right) \tag{2.53}$$

$D_{g,i,j,k}$ ：拡散係数
$J^{x\pm}_{g,i,j,k}$ ：x方向±側境界面における平均中性子流
$J^{y\pm}_{g,i,j,k}$ ：y方向±側境界面における平均中性子流
$J^{z\pm}_{g,i,j,k}$ ：z方向±側境界面における平均中性子流

拡散計算においては、ボルツマン輸送方程式に対して球面調和関数展開を行うことで導出される P1 方程式に基づいた次式の拡散係数が一般に用いられる[11]。

$$D = \frac{1}{3(\Sigma_{\mathrm{t}} - \Sigma_{\mathrm{s1}})} = \frac{1}{3\Sigma_{\mathrm{tr}}} \tag{2.54}$$

Σ_{tr} ：巨視的輸送面積

なお、上式においてエネルギー群の添字gは省略している。

ただし、開発したコードのS_N法計算では式(2.18)に示すように、巨視的全断面積が$\left(\Sigma_{\mathrm{t},g,i,j,k} - \alpha/v_g\right)$と補正されるような計算をしているため、拡散係数も同様に次式のように補正することとした。

$$D = \frac{1}{3\left(\Sigma_{\mathrm{tr}} - \frac{\alpha}{v}\right)} \tag{2.55}$$

2.3.2 中性子流補正係数

拡散加速法では、詳細計算で得られた正味の中性子流を再現する必要がある(空間均質化・エネルギー群縮約を行う場合は、反応率の保存も考慮する必要がある)。しかし、式(2.55)で定義した拡散係数を用いた拡散計算では、詳細計算の正味の中性子流を再現することができない。そこで拡散加速法では、拡散計算における中性子流に補正項を加えることで正味の中性子流を再現する。本項ではその際用いられる中性子流補正係数の計算方法について述べる。

まず、S_N法による詳細計算において、正味の中性子流は以下の式により求められる[1]。

$$J^{x-}_{g,i,j,k} = 4\pi \sum_{d=0}^{ND} w_d \psi^{x-}_{g,d,i,j,k} \mu_d \tag{2.56}$$

$$J^{x+}_{g,i,j,k} = 4\pi \sum_{d=0}^{ND} w_d \psi^{x+}_{g,d,i,j,k} \mu_d \tag{2.57}$$

$$J^{y-}_{g,i,j,k} = 4\pi \sum_{d=0}^{ND} w_d \psi^{y-}_{g,d,i,j,k} \eta_d \tag{2.58}$$

$$J^{y+}_{g,i,j,k} = 4\pi \sum_{d=0}^{ND} w_d \psi^{y+}_{g,d,i,j,k} \eta_d \tag{2.59}$$

$$J^{z-}_{g,i,j,k} = 4\pi \sum_{d=0}^{ND} w_d \psi^{z-}_{g,d,i,j,k} \xi_d \tag{2.60}$$

$$J^{z+}_{g,i,j,k} = 4\pi \sum_{d=0}^{ND} w_d \psi^{z+}_{g,d,i,j,k} \xi_d \tag{2.61}$$

ここで、詳細計算で得られた正味の中性子流$J_{g,i,j,k}^{\mathrm{fine},x+}$と拡散加速計算における正味の中性子流$J_{g,i,j,k}^{\mathrm{diff},x+}$が一致するように、以下のような補正を考える。

$$J_{g,i,j,k}^{\mathrm{diff},x+} = J_{g,i,j,k}^{\mathrm{fine},x+} = J_{g,i,j,k}^{\mathrm{FD},x+} + (\text{補正項})$$
$$= \frac{2D_{g,i,j,k}D_{g,i+1,j,k}}{D_{g,i,j,k}\Delta x_{i+1} + D_{g,i+1,j,k}\Delta x_i}\left(\phi_{i+1,j,k,g} - \phi_{g,i,j,k}\right) + D_{cor,g,i,j,k}^{x+}\left(\phi_{g,i+1,j,k} + \phi_{i,j,k,g}\right) \tag{2.62}$$

上式において、$J_{g,i,j,k}^{\mathrm{FD},x+}$は拡散理論に基づく有限差分式で得られる正味の中性子流、すなわち式(2.49)に相当する。上式右辺第 2 項の$D_{\mathrm{cor},g,i,j,k}^{x+}$が中性子流補正係数と呼ばれる(拡散係数との混同に注意)。$D_{\mathrm{cor},g,i,j,k}^{x+}$は上式を変形して得られる次式により計算できる。他の境界面についても同様に求めた補正項$D_{\mathrm{cor},g,i,j,k}^{y+}, D_{\mathrm{cor},g,i,j,k}^{z+}$も併せて示す。

$$D_{\mathrm{cor},g,i,j,k}^{x+}$$
$$= \frac{1}{\left(\phi_{g,i,j,k} + \phi_{i+1,j,k,g}\right)}\left\{J_{g,i,j,k}^{\mathrm{fine},x+} + \frac{2D_{g,i,j,k}D_{g,i+1,j,k}}{D_{g,i,j,k}\Delta x_{i+1} + D_{g,i+1,j,k}\Delta x_i}\left(\phi_{i+1,j,k,g} - \phi_{g,i,j,k}\right)\right\} \tag{2.63}$$

$$D_{\mathrm{cor},g,i,j,k}^{y+}$$
$$= \frac{1}{\left(\phi_{g,i,j,k} + \phi_{i,j+1,k,g}\right)}\left\{J_{g,i,j,k}^{\mathrm{fine},y+} + \frac{2D_{g,i,j,k}D_{g,i,j+1,k}}{D_{g,i,j,k}\Delta y_{j+1} + D_{g,i,j+1,k}\Delta y_j}\left(\phi_{i,j+1,k,g} - \phi_{g,i,j,k}\right)\right\} \tag{2.64}$$

$$D_{\mathrm{cor},g,i,j,k}^{z+}$$
$$= \frac{1}{\left(\phi_{g,i,j,k} + \phi_{i,j,k+1,g}\right)}\left\{J_{g,i,j,k}^{\mathrm{fine},z+} + \frac{2D_{g,i,j,k}D_{g,i,j,k+1}}{D_{g,i,j,k}\Delta z_{k+1} + D_{g,i,j,k+1}\Delta z_k}\left(\phi_{i,j,k+1,g} - \phi_{g,i,j,k}\right)\right\} \tag{2.65}$$

2.3.3 拡散加速計算

式(2.62)–(2.65)と同様にして各方向のメッシュ境界面について求めた、拡散加速計算における正味の中性子流$J^{\mathrm{diff},x\pm}_{g,i,j,k}, J^{\mathrm{diff},y\pm}_{g,i,j,k}, J^{\mathrm{diff},z\pm}_{g,i,j,k}$と中性子流補正係数$D^{x\pm}_{\mathrm{cor},g,i,j,k}, D^{y\pm}_{\mathrm{cor},g,i,j,k}, D^{z\pm}_{\mathrm{cor},g,i,j,k}$を式(2.47)に代入することで、次の 7 点階差式を得る。

$$\begin{aligned} &A^{x-}_{g,i,j,k}\phi_{g,i-1,j,k} + A^{y-}_{g,i,j,k}\phi_{g,i,j-1,k} + A^{z-}_{g,i,j,k}\phi_{g,i,j,k-1} + A^{0}_{g,i,j,k}\phi_{g,i,j,k} + \\ &A^{x+}_{g,i,j,k}\phi_{g,i+1,j,k} + A^{y+}_{g,i,j,k}\phi_{g,i,j+1,k} + A^{z+}_{g,i,j,k}\phi_{g,i,j,k+1} \\ &= Q_{g,i,j,k} \end{aligned} \tag{2.66}$$

$$Q_{g,i,j,k} = \frac{\alpha}{v_g}\phi_{g,i,j,k} + \sum_{\substack{g'=1 \\ g'\neq g}}^{NG} \Sigma_{\mathrm{s0},g'\to g,i,j,k}\phi_{g',i,j,k} \tag{2.67}$$

ここで、各係数Aは通常の拡散計算とは異なり、以下のように表される。

$$A^{x-}_{g,i,j,k} = -\frac{2D_{g,i-1,j,k}D_{g,i,j,k}}{D_{g,i-1,j,k}\Delta x_i + D_{g,i,j,k}\Delta x_{i-1}}\left(\phi_{g,i,j,k} - \phi_{i-1,j,k,g}\right) - \frac{D^{x-}_{\mathrm{cor},g,i,j,k}}{\Delta x_i} \tag{2.68}$$

$$A^{x-}_{g,i,j,k} = -\frac{2D_{g,i,j,k}D_{g,i+1,j,k}}{D_{g,i,j,k}\Delta x_{i+1} + D_{g,i+1,j,k}\Delta x_i}\left(\phi_{i+1,j,k,g} - \phi_{g,i,j,k}\right) + \frac{D^{x+}_{\mathrm{cor},g,i,j,k}}{\Delta x_i} \tag{2.69}$$

$$A^{y-}_{g,i,j,k} = -\frac{2D_{g,i,j-1,k}D_{g,i,j,k}}{D_{g,i,j-1,k}\Delta y_j + D_{g,i,j,k}\Delta y_{j-1}}\left(\phi_{g,i,j,k} - \phi_{i,j-1,k,g}\right) - \frac{D^{y-}_{\mathrm{cor},g,i,j,k}}{\Delta y_j} \tag{2.70}$$

$$A^{y+}_{g,i,j,k} = -\frac{2D_{g,i,j,k}D_{g,i,j+1,k}}{D_{g,i,j,k}\Delta y_{j+1} + D_{g,i,j+1,k}\Delta y_j}\left(\phi_{i,j+1,k,g} - \phi_{g,i,j,k}\right) + \frac{D^{y+}_{\mathrm{cor},g,i,j,k}}{\Delta y_j} \tag{2.71}$$

$$A^{z-}_{g,i,j,k} = -\frac{2D_{g,i,j,k-1}D_{g,i,j,k}}{D_{g,i,j,k-1}\Delta z_k + D_{g,i,j,k}\Delta z_{k-1}}\left(\phi_{g,i,j,k} - \phi_{i,j,k-1,g}\right) - \frac{D^{z-}_{\mathrm{cor},g,i,j,k}}{\Delta z_k} \tag{2.72}$$

$$A^{z+}_{g,i,j,k} = -\frac{2D_{g,i,j,k}D_{g,i,j,k+1}}{D_{g,i,j,k}\Delta z_{k+1} + D_{g,i,j,k+1}\Delta z_k}\left(\phi_{i,j,k+1,g} - \phi_{g,i,j,k}\right) + \frac{D^{z+}_{\mathrm{cor},g,i,j,k}}{\Delta z_k} \tag{2.73}$$

$$\begin{aligned} A^{0}_{g,i,j,k} = \Sigma_{\mathrm{r},g,i,j,k} &- \left(A^{x-}_{g,i,j,k} + A^{x-}_{g,i,j,k} + A^{y-}_{g,i,j,k} + A^{y+}_{g,i,j,k} + A^{z-}_{g,i,j,k} + A^{z+}_{g,i,j,k}\right) \\ &+ \frac{2\left(D^{x+}_{\mathrm{cor},g,i,j,k} - D^{x-}_{\mathrm{cor},g,i,j,k}\right)}{\Delta x_i} + \frac{2\left(D^{y+}_{\mathrm{cor},g,i,j,k} - D^{y-}_{\mathrm{cor},g,i,j,k}\right)}{\Delta y_j} \\ &+ \frac{2\left(D^{z+}_{\mathrm{cor},g,i,j,k} - D^{z-}_{\mathrm{cor},g,i,j,k}\right)}{\Delta z_k} \end{aligned} \tag{2.74}$$

$\Sigma_{\mathrm{r},g}$　：g群の巨視的除去断面積 $(\Sigma_{\mathrm{r},g} = \Sigma_{\mathrm{t},g} - \Sigma_{\mathrm{s0},g\to g})$

拡散加速法では、式(2.66)のα固有値方程式を解くことで得られる、詳細計算における正味の中性子流と反応率を再現した全中性子束を詳細計算に反映することで、詳細計算の収束を加速させる。α固有値計算の外部反復を対象とした拡散加速計算の具体的な手順を以下に示す。

1. 詳細計算のn回目の transport sweep が完了した後、詳細計算の$\phi_0^{0\,\mathrm{fine},(n)}$及び正味の中性子流$J^{\mathrm{fine},(n)}$、$\alpha^{\mathrm{fine},(n-1)}$を用いて、中性子流補正係数$D_{\mathrm{cor}}$及び係数$A$を計算する。ただし、中性子流$J^{\mathrm{fine},(n)}$は、transport sweep の際に式(2.56)–(2.61)により計算しておく必要がある。ここで、上付き文字のfineは詳細計算の変数であることを表す。
2. 詳細計算n回目の外部反復の全中性子束$\phi_0^{0,(0)}$を、拡散加速計算の全中性子束初期値$\phi^{\mathrm{diff},(0)}$とする。αの初期値$\alpha^{\mathrm{diff},(0)}$は次式により与える。ここで、上付き文字のdiffは拡散加速計算の変数であることを表す。

$$\alpha^{\mathrm{diff},(n')} = \frac{1}{\sum_{g=1}^{NG}\sum_{k=1}^{NZ}\sum_{j=1}^{NY}\sum_{i=1}^{NX}\frac{\phi_{i,j,k,g}^{\mathrm{diff},(n')}}{v_g}\Delta x_i \Delta y_j \Delta z_k} \tag{2.75}$$

3. n'回目の外部反復開始(拡散加速計算の外部反復回数をn'で示すこととする)。
4. $\alpha^{\mathrm{diff},(n')}$と$\phi_{g,i,j,k}^{\mathrm{diff},(n')}$を用いて、式(2.67)により各エネルギー群の各メッシュに対し$Q_{i,j,k,g}^{\mathrm{diff},(n')}$を計算する。
5. 連立方程式を各エネルギー群について解き、$\phi_{g,i,j,k}^{\mathrm{diff},(n'+1)}$を計算する。(内部反復計算)
6. 式(2.75)により$\alpha^{\mathrm{diff},(n'+1)}$を計算する。
7. $\phi_{g,i,j,k}^{\mathrm{diff}}$及び$\alpha^{\mathrm{diff}}$の収束判定基準を満足するまで、手順 3.から 6.を繰り返す。
8. 収束した全中性子束$\phi_{g,i,j,k}^{\mathrm{diff}}$を用いて次式のように詳細計算の中性子束を更新して、拡散加速計算を完了する。

$$\phi_{l}^{m\,\mathrm{fine},(n,\mathrm{after})}{}_{g,i,j,k} = \phi_{l}^{m\,\mathrm{fine},(n,\mathrm{before})}{}_{g,i,j,k}\frac{\phi_{g,i,j,k}^{\mathrm{diff}}}{\phi_{0}^{0\,\mathrm{fine}(n,\mathrm{before})}{}_{g,i,j,k}} \tag{2.76}$$

2.4 delta-tracking 法による数値不安定性の改善

本研究で開発した計算コードにおいて、アルミニウムのような密度が小さい、すなわち巨視的全断面積が小さい材質が計算体系に含まれる場合に、拡散加速計算が不安定化または発散しやすい問題があることがわかった。より具体的には、全断面積が小さい場合にS_N法の詳細計算に中$\left(\Sigma_{\mathrm{t},g,i,j,k}-\alpha/v_g\right)<0$となる場合があり、これにより生じる負の角度中性子束が数値不安定性をもたらす。そこで本研究では、本来モンテカルロ計算において計算を効率化するための手法である delta-tracking 法[4]を、数値不安定性の改善のために用いることとした。以下では、その具体的な手法について述べる。

delta-tracking 法では、次式のように各次の自群散乱断面積に仮想的な散乱断面積$\Sigma_{0,g}$を加える。

$$\Sigma^{*}_{\mathrm{s}l,g'\to g}=\Sigma_{\mathrm{s}l,g'\to g}+\Sigma_{0,g} \tag{2.77}$$

$\Sigma^{*}_{\mathrm{s}l,g'\to g}$ ：g'群からg群へのl次の補正された巨視的非等方散乱断面積

$\Sigma_{0,g}$ ：g群の仮想散乱断面積

これに対応して、巨視的全断面積を以下のように補正する。

$$\Sigma^{*}_{\mathrm{t},g}=\Sigma_{\mathrm{t},g}+\Sigma_{0,g} \tag{2.78}$$

$\Sigma^{*}_{\mathrm{t}l,g}$:g群の補正された巨視的全断面積

これらの補正の結果、元のα固有値方程式(2.79)は式(2.80)のように書き換えられる。

$$\boldsymbol{\Omega}_d\cdot\boldsymbol{\nabla}\psi_{g,d}+\left\{\Sigma_{t,g}-\frac{\alpha}{v_g}\right\}\psi_{g,d}=\sum_{l=0}^{NL}\frac{2l+1}{4\pi}\sum_{m=-l}^{l}\sum_{g'=1}^{NG}\Sigma_{\mathrm{s}l,g'\to g}\,\phi_l^m{}_g R_l^m(\boldsymbol{\Omega}_d) \tag{2.79}$$

$$\begin{aligned}\boldsymbol{\Omega}_d\cdot\boldsymbol{\nabla}\psi_{g,d}+\left\{\Sigma_{t,g}-\frac{\alpha}{v_g}+\Sigma_{0,g}\right\}\psi_{g,d}\\ =\sum_{l=0}^{NL}\frac{2l+1}{4\pi}\sum_{m=-l}^{l}\sum_{g'=1}^{NG}\left(\Sigma_{\mathrm{s}l,g'\to g}+\Sigma_{0,g}\delta_{gg'}\right)\phi_l^m{}_g R_l^m(\boldsymbol{\Omega}_d)\end{aligned} \tag{2.80}$$

$\delta_{gg'}$ ：クロネッカーのデルタ

このとき、式(2.28)より展開次数上限NLを十分に大きくとれば、式(2.80)の左辺の$\Sigma_{0,g}\psi_{g,d}$と右辺の仮想散乱源$\sum_{l=0}^{NL}\{(2l+1)/4\pi\}\sum_{m=-l}^{l}\sum_{g'=1}^{NG}\Sigma_{0,g}\delta_{gg'}\,\phi_{l,g}^m R_l^m(\boldsymbol{\Omega}_d)$は等しくなり、互いに打ち消す合うことがわかる。ゆえに、このように設定した仮想散乱は、散乱前後で中性子エネルギーの飛行方向が変化しない散乱であるといえる。

$$\psi(\boldsymbol{r},\boldsymbol{\Omega},E)=\sum_{l=0}^{NL}\frac{2l+1}{4\pi}\sum_{m=-l}^{l}\phi_l^m(\boldsymbol{r},E)R_l^m(\boldsymbol{\Omega}) \tag{2.28}$$

再掲

ここで、例えば計算体系のαの推定値をα_{est}として、仮想散乱断面積を$\Sigma_{0,g}=\alpha_{\mathrm{est}}/v_g$のように設定すれば、$\left(\Sigma_{\mathrm{t},g,i,j,k}-\alpha/v_g\right)$の項は$\left(\Sigma_{\mathrm{t},g,i,j,k}-\alpha/v_g+\alpha_{\mathrm{est}}/v_g\right)$のように補正され、負の値をとりにくくなることから、数値不安定性を改善することができる。

2.5　本章のまとめ

本章では、S_N法に基づくα固有値計算理論について述べた。

2.2 節では、S_N法に基づくα固有値方程式の数値解法について述べた。まず、2.2.1 項で輸送理論に基づくα固有値方程式を示した。そして、2.2.2 項、2.2.3 項、2.2.4 項でエネルギー、飛行方向、空間について離散化を行い、S_N法に基づく離散化されたα固有値方程式を導出した。また、2.2.5 項では本研究で取り扱う真空境界条件について、2.2.6 項では実球面調和関数展開を利用した非等方散乱成分の取り扱いについて説明した。2.2.7 項では角度分点セットについて述べた。本研究では、少ない飛行方向分割数で高精度な球面調和関数の積分計算が可能である特徴を持つ icosahedral 分点を利用するが、方向余弦が 0 とならないように分点をいくらか回転する必要があり、それに伴い完全反射境界条件の適用が困難であることに注意しなければならない。そして 2.2.8 項では、導出したα固有値方程式の最小固有値に対応する即発中性子減衰定数αを、逆べき乗法に相当する反復解法により得る手順を述べた。2.2.9 項では、反復解法において必要な、与えられた中性子源$Q_{g,d,i,j,k}$に対し各飛行方向について角度中性子束$\psi_{g,d,i,j,k}$を更新する計算である、transport sweep について述べた。transport sweep では、あるメッシュについての計算は隣接メッシュからの流入角度中性子束に依存するため、飛行方向に応じて計算順序を変える必要がある。

2.3 節ではS_N法の収束を加速するための拡散加速法について述べた。2.3.1 項では、拡散加速法の基本となる拡散理論に基づくα固有値方程式を示した。拡散加速法では詳細計算で得られた正味の中性子流を再現する必要があることから、2.3.2 項では拡散加速計算において詳細計算(S_N法計算)で得られる正味の中性子流を再現するための中性子流補正係数D_{cor}について述べた。2.3.3 項では拡散加速計算で解くべきα固有値方程式を導出し、正味の中性子流を再現した全中性子束を計算し、詳細計算に反映する手順について述べた。

2.4 節では、delta-tracking 法による数値不安定性の改善について述べた。本研究で開発した計算コードにおいて、巨視的全断面積が小さい材質が計算体系に含まれる場合に$\left(\Sigma_{\mathrm{t},g,i,j,k}-\alpha/v_g\right)$の項が負となることで、拡散加速計算が不安定化または発散しやすい問題があることがわかった。そこで、仮想散乱源を導入することで$\left(\Sigma_{\mathrm{t},g,i,j,k}-\alpha/v_g\right)$の項を$\left(\Sigma_{\mathrm{t},g,i,j,k}-\alpha/v_g+\alpha_{\mathrm{est}}/v_g\right)$のように補正し、負の値をとりにくくすることで数値不安定性の改善を図った。

2.6 参考文献

[1] D. G. CACUCI, Ed., *Handbook of nuclear engineering*, Vol. 2, Springer, New York ; London (2010).

[2] K. KOBAYASHI, *原子炉物理*, コロナ社, Tokyo (1996).

[3] N. Z. CHO, C. J. PARK, "A Comparison of Coarse Mesh Rebalance (CMR) and Coarse Mesh Finite Difference (CMFD) Acceleration Methods for the Neutron Transport Calculations," *Proc. M&C 2003*, Gatlinburg, Tennessee, April 6–11, 2003, American Nuclear Society (2003).

[4] J. LEPPÄNEN, "Performance of Woodcock delta-tracking in lattice physics applications using the Serpent Monte Carlo reactor physics burnup calculation code," *Annals of Nuclear Energy*, 37 5, 715 (2010); https://doi.org/10.1016/j.anucene.2010.01.011.

[5] T. ENDO, A. NOGUCHI, A. YAMAMOTO, et al., "Perturbation-Theory-Based Sensitivity Analysis of Prompt Neutron Decay Constant for Water-Only System", *Transactions of American Nuclear Society*, 124, pp.184-187 (2021).

[6] R. E. ALCOUFFE, R. S. BAKER, J. A. TURNER, et al., "PARTISN MANUAL," LA-UR-02-5633, Los Alamos National Laboratory (2002).

[7] G. LONGONI, A. HAGHIGHAT, J. BROWN, et al., "Investigation of new quadrature sets for discrete ordinates method with application of non-conventional problems," *Proc. 2001 ANS Summer Meeting*, (2001).

[8] C. AHRENS and G. BEYLKIN, "Rotationally invariant quadratures for the sphere," *Proc. R. Soc. A*., 465 2110, 3103 (2009); https://doi.org/10.1098/rspa.2009.0104.

[9] K. MANALO, C. D. AHRENS, and G. SJODEN, "Advanced quadratures for three-dimensional discrete ordinate transport simulations: A comparative study," *Annals of Nuclear Energy*, 81, 196 (2015); https://doi.org/10.1016/j.anucene.2015.02.032.

[10] T. ENDO, "Study and Visualization of Icosahedral Quadrature," ResearchGate; https://www.researchgate.net (accessed Dec 6, 2023).

[11] A. YAMAMOTO, "拡散方程式の数値解法の基礎," *第 36 回炉物理夏期セミナーテキスト*, pp.47–80 (2004).

第3章　GPGPU を用いたS_N法に基づくα固有値計算コードの開発

3.1　本章の概要

本章では、GPGPU を用いたS_N法に基づくα固有値計算コードの開発について述べる。3.2 節では、GPU を科学技術計算などに応用する技術である GPGPU の概要を説明する。3.3 節では、GPGPU を用いた核計算コード開発において一般に課題となる事柄について述べる。3.4 節及び 3.5 節では、S_N法に基づくα固有値計算及びα固有値拡散加速計算を GPGPU により実装する手法について、それぞれ具体的に述べる。

3.2　GPGPU の概要

GPGPU(General-Purpose Computing on Graphics Processing Units)は、コンピュータに搭載される画像処理装置である GPU(Graphics Processing Unit)を科学技術計算などの目的に応用する技術である。

並列処理可能なスレッド数が数十程度と少ない代わりに、個々のスレッドを高速に処理できるよう設計されている CPU に対して、GPU は個々のスレッドの処理が低速である代わりに、数千のスレッドを並列処理できるように特化した設計がなされている。具体的には、GPU は図 3.1 のように、データキャッシングやフロー制御ではなく、浮動小数点演算などのデータ処理のためにより多くのトランジスタ(半導体素子)が演算コアとして使用されるよう設計されている。ゆえに、GPU では単純で高い並列性をもつ計算において高い性能が得られる一方で、条件分岐が多く並列性が低い計算は不得手であるという特徴をもつ[1]。

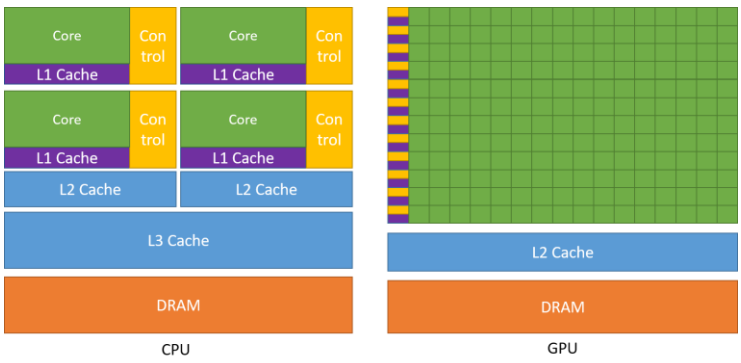


図 3.1 CPU と GPU の設計の違い[1]

ここで、図 3.1 中の各種ユニットは以下のような役割を持つ。

- Core (コア):
  四則演算や論理演算の処理を行う。
- Cache (キャッシュ):
  使用頻度の高い命令やデータを格納する。小容量だが高帯域・低遅延であり、キャッシュを利用して低帯域・高遅延な外部メモリへのアクセスを減らすことで高速な動作を実現する。CPU や GPU は L1, L2 といったように複数レベルのキャッシュを持つことが多く、L1 キャッシュが最も高速である。
- Control (コントローラ):
  与えられた命令に従ってコアやキャッシュを制御する。
- DRAM (Dynamic Random Access Memory):
  大容量な外部メモリで、CPU や GPU のメインメモリ。低価格だが、高遅延・低帯域。

GPGPU 向けのプログラミング環境には、CUDA[2]、OpenCL[3]、OpenACC[4]、OpenMP[5]などが存在するが、本研究ではそれらの中でも情報が豊富な CUDA を用いることとした。なお本論文において単に CUDA と記した場合は通常、C++の拡張として利用できる CUDA C++を指すこととする。

次項からは、CUDA を利用したプログラミングをするにあたって知っておくべき事柄について述べる。3.2.1 項では CUDA の大まかなプログラミングモデルについて、3.2.2 項及び 3.2.3 項では CUDA の特徴的なスレッド及びメモリの階層構造についてそれぞれ述べる。

3.2.1 CUDA プログラミングモデル

まず、本論文において CUDA について述べる際は、本研究で使用した GPU である NVIDIA RTX 4090 が対応する Computation Capability[1] 8.9 を前提とする。Computation Capability により仕様が異なる場合は必要に応じて述べるが、全てを網羅しているわけではなく、新バージョンで仕様が変わる場合もあることに注意されたい。正確な情報を得るには、使用する GPU が対応する Computation Capability に注意し、文献[1]などの NVIDIA 社による最新の CUDA 公式ドキュメントを参照されたい。以降では、本研究で開発したコードについて理解するために必要であると考えられる、CUDA の仕様の一部について述べる。

CUDA C++では、GPU 上で実行したいコードを、カーネル(kernel)と呼ばれる C++関数として定義する。通常の C++関数が 1 度呼び出されると 1 回だけ実行されるのに対し、kernel が 1 度呼び出されると、N個のスレッドが同じ kernel を並列実行する。各スレッドには組み込み変数を介して一意のスレッド ID が与えられ、各スレッドはスレッド ID をもとに操作すべきメモリアドレスなどを得ることとなる[1]。

[1] GPU ハードウェアがサポートする機能を識別するバージョン番号のこと。

例として、要素数がNのベクトルA, Bを加算して結果をCに格納するコードを以下に示す。このコードでは、N個のスレッドそれぞれがベクトルA, Bの要素 1 ペアの計算を実行する。

Program 3.1 要素数NのベクトルA, Bを加算する kernel の例 ([1]より一部改変)

注：実際には次項以降で説明する block や grid の考え方を踏まえて記述する必要がある

```
// カーネルの定義
__global__ void VecAdd(float* A, float* B, float* C) {
  int i = threadIdx.x; // 各スレッドについて一意の ID
  C[i] = A[i] + B[i];
}

int main() {
  ...
  // カーネルの呼び出し N スレッドで並列実行
  VecAdd<<<1, N>>>(A, B, C);
  ...
}
```

CUDA プログラミングモデルでは、C++プログラムを実行するホスト(host)に対して、物理的に異なるデバイス(device)上で CUDA スレッドが実行されることを前提としている。通常、host は CPU を、device は GPU を指す。また、host と device がそれぞれホストメモリ(host memory)とデバイスメモリ(device memory)と呼ばれる別個のメモリ空間を持つことも前提とされる[1]。ゆえに、host 側の計算結果を device 側の計算で利用する場合は、データを事前に host memory から device memory へ転送する必要があり、その逆も同様である。CUDA プログラミングモデルに基づいたプログラムのイメージを下図に示す。

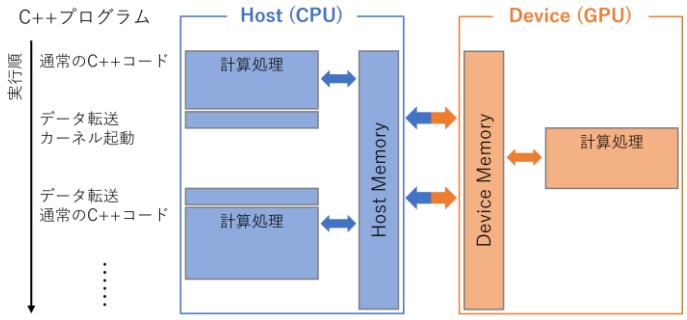


図 3.2 CUDA プログラミングモデル

ただし、host memory・device memory 間の帯域幅は十数～数十 GB/s であり、device memory・device 間(つまり GPU 内でのデータ転送)の帯域幅数百 GB/s に大きく劣る。また、host・device 間のデータ転送は大きなオーバーヘッドを伴う。したがって、host・device 間のデータ転送をなるべく少なくするとともに、host・device 間のデータ転送では小さなデータを 1 つ 1 つ

転送するのではなく、大きなデータとしてまとめて 1 回で転送することが望ましい[6]。

3.2.2 CUDA におけるスレッド階層と SIMT アーキテクチャ

CUDA では、図 3.3 のようにスレッド(thread)に階層構造が設けられている。最大 1024 個の thread が thread block(以降、単に block と表記)を形成し、さらに複数の block が grid を形成する。また、block は 1 次元から 3 次元までの構造をもつことができ、例えば block を 2 次元として行列あるいは部分行列の行インデックス､列インデックスをそれぞれ 1 次元目、2 次元目に割り当てることで取り扱いを簡単にすることができる。同様に、grid も 1 次元から 3 次元までの構造をもつことができる。block 中の thread ID 及び、grid 中の block ID はそれぞれ、組み込み変数 `threadIdx` 及び `blockIdx` を用いて知ることができる。また、block 及び grid のサイズを知るには `blockDim` 及び `gridDim` を利用できる。例えば、2 次元 block 中の thread ID は`(blockDim.x * threadIdx.y + threadIdx.x)`と計算できる。

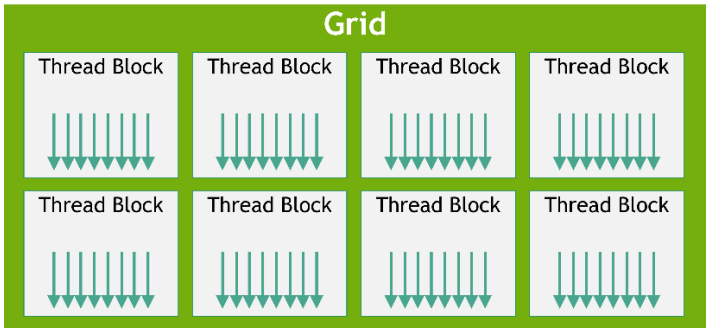


図 3.3 CUDA スレッドの階層構造 (矢印はスレッドを表す) [1]

kernel が起動されると、block は GPU 上に数十～百数十基搭載されている streaming multiprocessor(SM)に分配され、実行される。SM はいくつかの block を同時に実行することができ、SM 上の block の実行が終了して空きができると、未起動の block がその SM 上で起動される。通常、GPU の性能を十分に利用するには grid 中の block の数が GPU 上の SM の数を超える必要がある。また、基本的に各 block は独立して実行できる必要があり、プログラマは block がどの SM に割り当てられるか、どの順序で実行されるかを指定することはできない[1]。

さらに、block 中の thread を 32 個まとめたものは warp と呼ばれる。block の形状によらず、block 中 0 番目の thread から順に連続した 32 thread が 1 warp に割り当てられる。そして、それぞれの warp は SM 中にいくつか搭載されている warp scheduler によって実行スケジュールを設定される。warp は一度に、warp 内の thread で共通の命令(instruction)を同時に実行する。逆に言えば、warp 内の thread は異なる命令を同時に実行することはできない。ゆえに、`if` 文などの条件分岐において warp 内の全 thread が同じ条件に分岐しない場合、

warp はそれぞれの分岐を実行するときにその分岐を通らない thread を無効化する(図 3.4)[1]。warp 内の分岐が増えていくと動作せず休眠するスレッドが増え、スループットが低下する。これは warp divergence と呼ばれる。この分岐は warp 内でのみ生じるもので、異なる warp に影響することはない。

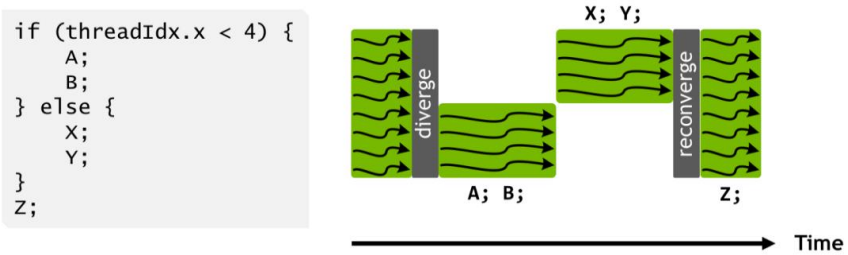


図 3.4 warp 内分岐がある場合の thread のスケジューリングの例
(Volta アーキテクチャより前) [7]

このような「一つの命令を、複数のスレッドが、同時に実行する」設計は、SIMT(Single-Instruction, Multiple-Thread)アーキテクチャと呼ばれる[1]。これに対し、CPU のアーキテクチャは基本的に SIMD(Single-Instruction, Multiple-Data)である。なお、NVIDIA Volta アーキテクチャ(Computation Capability 7.x)より前のアーキテクチャでは、命令ごとに warp が同期していることが保証されていたが、thread スケジューリングの柔軟化のために Volta アーキテクチャ以降で導入された independent thread scheduling により、warp 同期性は暗黙的には保証されない(図 3.5)ことに注意する必要がある。ただし、明示的に warp を同期することは可能である。詳しくは NVIDIA 社の公式ドキュメント[1],[7]を参照されたい。

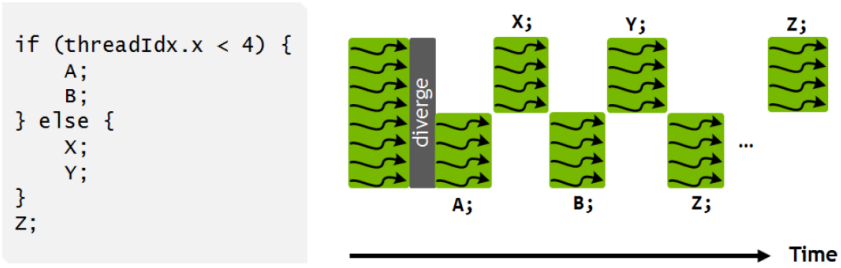


図 3.5 warp 内分岐がある場合の thread のスケジューリングの例
(Volta アーキテクチャ以降) [7]
warp 内で Z;が同時実行される暗黙的な保証はない

CPU 上のスレッドは一般に重量級であり、実行する thread を切り替えるのには時間とコストを要する。一方で、GPU 上の thread は非常に軽量であり、短時間・低コストで実行する thread を warp 単位で切り替えることができる。CUDA では、図 3.6 に示すようにある warp の命令[2]を発行した後、その実行が完了するまでのレイテンシ(latency, 遅延)の間に、実行の準備が完了した warp(eligible warp)に切り替え、命令を次々に発行する。これにより命令のレイテンシを隠蔽(latency hiding)して大きなスループットを得ることが CUDA の基本的な考え方である。なお、CPU ではレイテンシを最小化することが重視される[6]。

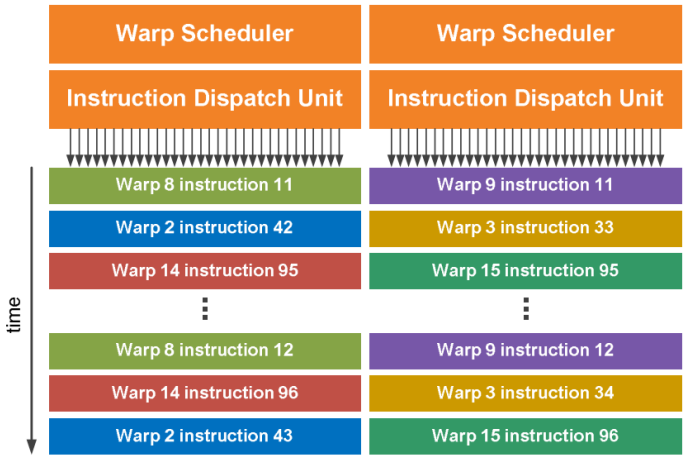


図 3.6 CUDA における warp の命令発行のイメージ[8]

また、命令の実行完了待ちになっている warp は stalled warp と呼ばれる。レイテンシの大きな命令を多く発行すると、stalled warp が増加し、eligible warp の数が SM 中の warp scheduler の数を下回ってしまう。こうなると、warp scheduler は次に命令を発行すべき warp を選択できないため、レイテンシを隠蔽しきれなくなり、スループットが低下する[9]。ゆえに、後述する global memory アクセスのようなレイテンシが大きな命令を多く発行することを避け、かつ SM にできるだけ多くの warp が駐在できるようにすることが望ましい。

実行時、各 SM に一度に駐在できる warp は active warp と呼ばれ、その数の理論値は theoretical occupancy(理論的占有率、デバイスの制限に対する割合で表されることもある)と呼ばれる。theoretical occupancy は block のサイズ、後述する shared memory や register の使用量により制限される。十分な数の warp があり、warp scheduler が適切に動作すれば、active warp の数は theoretical occupancy に近い値になる[9]。warp の各状態と制限の関係を図 3.7 に示す。

[2] ここでいう命令(instruction)とは、あるデータをメモリから register へロードする命令や、register のデータを使って演算する命令などを指す。プログラム上の 1 行は大抵の場合、複数個の命令となる。

図 3.7 warp の各状態と制限の関係[9]

warp についてはすでに述べたが、block と warp はそれぞれに属する thread の実行を同期することができる[1]。例えば、block 内の thread 間で計算に依存性があるような場合に、次項で述べる share memory を利用して block 内でデータを共有し、実行を同期することで thread 間の連携が可能となる。逆に、block 内で実行を同期していないと、必要な計算が完了していない可能性がある[1]。なお、grid は kernel の起動時と終了時に同期するのみであり、先述したように block は基本的に独立して実行できる必要がある。

3.2.3　CUDA におけるメモリ階層及びメモリアクセス

CUDA プログラミングにおいては、スレッドが階層構造を持つように、メモリも図 3.8 のような階層構造を持つ。thread ごとにレジスタ(register)と local memory が存在し、block ごとに shared memory が存在する。また、GPU 上の全ての thread からアクセス可能な global memory が存在する[1]。

この階層構造においては、ある thread の register に対して異なる thread からアクセスすることはできず、同様にある block の shared memory に対して異なる block からアクセスすることはできない。ただし、Compute Capability 9.0 では thread block cluster と呼ばれる新たな階層が導入され、cluster 内の block は互いの shared memory を利用できるようになった(図 3.8)[1]。しかし、本研究で利用した GPU はこれに対応していないため利用しない。

一般に、local memory を除き図 3.8 において上のメモリであるほど高帯域幅・低レイテンシであり、register が最も高速である。一方で、register や shared memory は SM1 基あたりの容量が数十～数百 KB と小さく、global memory は GPU1 基につき数 GB～数十 GB と大きな容量を持つ。

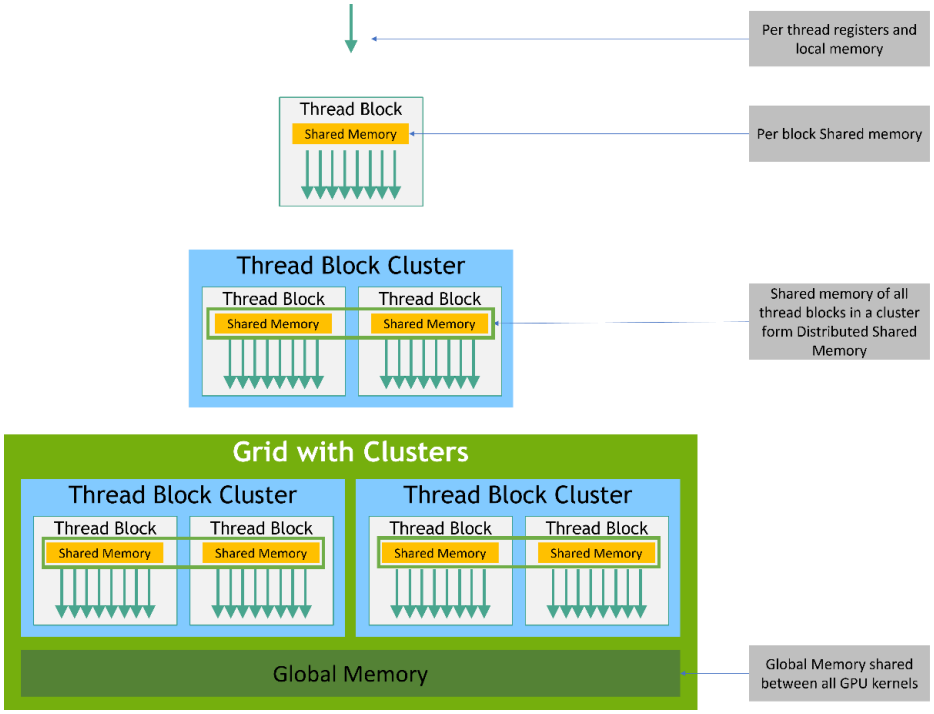


図 3.8 CUDA におけるメモリ階層[1]

3.2.2 項と本項のここまでの内容を簡単にまとめると、階層構造上で近い thread 同士であるほど、より緊密な実行の同期や高速なデータ通信ができるといえる。CUDA を利用したプログラミングでは、特に global memory と shared memory の利用の仕方が重要となる。以下では、各メモリの特徴について述べる。

a)　global memory

global memory は device 全体で共有されるオフチップメモリである。cudaMalloc()関数などにより確保することができ、host とのやりとりが必要となるデータは基本的に global memory に格納することとなる。オフチップメモリであるため、オンチップメモリである shared memory や register に比べ大容量ではあるが低帯域幅(最大で数百 GB/s)・高レイテンシである。そのため、CUDA を利用したプログラミングにおいては、global memory に対する冗長なアクセスを減らすことや、アクセスを合体させる、すなわち coalesced access(コアレスドアクセス)が性能を引き出すために非常に重要とされる[1],[6]。

global memory へのアクセスは必ず、32 byte の倍数となる位置から 32 byte ごとに処理される。以降、この処理を 32 byte トランザクションと表記する。warp 内の thread が global memory へ同時にアクセスするとき、その処理に必要なだけの 32 byte トランザクションが実行される。単純な例として、float 型のような 4 byte を 1 要素とする配列に対し、warp の全 32 thread が連続する 32 要素に同時にアクセスする場合を考える。このとき、メモリが 32 byte の倍数にアライメントされていれば、必要となるのは 4 つの 32 byte トランザクションとなる(図 3.9)。このとき、各トランザクションにつき$32\ \text{byte} = 4\ \text{byte} \times 8$要素で 8 要素分のアクセスが合体されて(coalesced)いることから、coalesced access と呼ばれる[6]。

図 3.9 アライメントされた適切な coalesced access の例[6]

一方で、32 byte の倍数にアライメントされていない場合は図 3.10 のように 5 つの 32 byte トランザクションが必要となる。この場合、同じ量のデータにアクセスするために5/4倍のトランザクションを必要とすることとなり、性能が低下する。単純に考えるとスループットが4/5倍に低下するように思われるが、この例のように隣接する warp によりフェッチされたキャッシュラインが再利用できる場合は、性能低下は9/10程度にとどまることがある[6]。

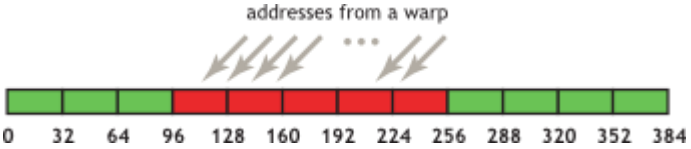


図 3.10 ミスアライメントされたメモリアクセスの例[6]

なお、global memory へのアクセスは通常 L1, L2 キャッシュによりキャッシュされる。また、L2 キャッシュへのリクエストの粒度は 128 byte であることから、1 度のアクセスにつき 4 つの 32 byte トランザクションが発生することが理想的とされる。

図 3.9 及び図 3.10 の例においていくつかの thread がアクセスに参加しなかった場合も、32 byte を単位としてアクセスが行われることに変わりはない。ゆえに、下図のようなストライド幅が 2 となるストライドアクセスの場合、1 つの 32 byte トランザクションにつき 16 byte 分のデータにのみアクセスすることとなり、データ転送効率は$1/2$に低下することとなる。ストライド幅が増していくとさらに効率は低下し、スループットも低下する。具体的には、ストライド幅が 2 のとき実効帯域幅は適切な場合の$1/2$以下になり、さらにストライド幅が増せば$1/10$以下になり得る[6]。

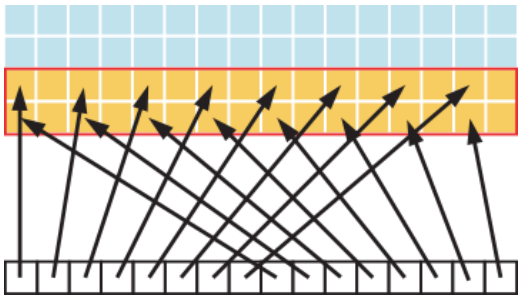


図 3.11 ストライド幅 2 の global memory アクセス[6]

よって、global memory に対するストライドアクセスは可能な限り避けるべきである。これを回避するには、例えば後述する shared memory を利用することができる。

b) shared memory

shared memory は block 内で共有されるオンチップメモリである。オンチップメモリであるため、global memory や local memory に比べはるかに高帯域幅(最大で数 TB/s)・低レイテンシである[6]。ただし、SM1 基、block あたりの容量に制限があることに注意する必要がある。

shared memory は、block 内で何度もアクセスされるようなデータに対して利用すると効果的である。一方で、global memory から shared memory へデータをコピーした後、それに一度しかアクセスしないようなデータに対して利用することは効果的ではない。ただし、後述するがストライドアクセスに対しては効果的な場合もある。shared memory はプログラマが明示的に利用できるキャッシュメモリであると考えると理解しやすいと思われる。

ただし、shared memory を利用する際には bank conflict (bank の競合)をなるべく発生させないよう注意が必要である。shared memory は図 3.12 のように 4 byte 刻みで 32 個の bank に分かれており、warp 中の複数 thread による、異なる bank に属するアドレスに対するメモリアクセスは同時に処理することができる。なお、アドレスが連続である必要はない。したがって、最大で 32 個の異なる bank にある 128 byte のデータへのアクセスを同時に処理することができる。ただし、warp 中の 2 つ以上 thread が要求したアクセスが同じ bank で異なるアドレスを指しているような場合には bank conflict が発生し、競合しないよう処理が分割される。分けられた処理の数nに応じてn-way bank conflict と呼称し、このときスループットはおよそ$1/n$に低下する[1],[6]。図 3.13 に、bank conflict が発生する場合と発生しない場合の例をいくつか示す。

Bank	float a[ ]			
0	0	32	64	···
1	1	33	65	···
2	2	34	66	···
3	3	35	67	···
4	4	36	68	···
5	5	37	69	···
6	6	38	70	···
7	7	39	71	···
8	8	40	72	···
9	9	41	73	···
10	10	42	74	···
11	11	43	75	···
12	12	44	76	···
13	13	45	77	···
14	14	46	78	···
15	15	47	79	···
16	16	48	80	···
17	17	49	81	···
18	18	50	82	···
19	19	51	83	···
20	20	52	84	···
21	21	53	85	···
22	22	54	86	···
23	23	55	87	···
24	24	56	88	···
25	25	57	89	···
26	26	58	90	···
27	27	59	91	···
28	28	60	92	···
29	29	61	93	···
30	30	62	94	···
31	31	63	95	···

図 3.12 `float` 型(4 byte)配列が格納されている場合の配列インデックスと bank の関係
(8 byte の `double` 型の場合、1 要素が 2 つの bank に対応する)

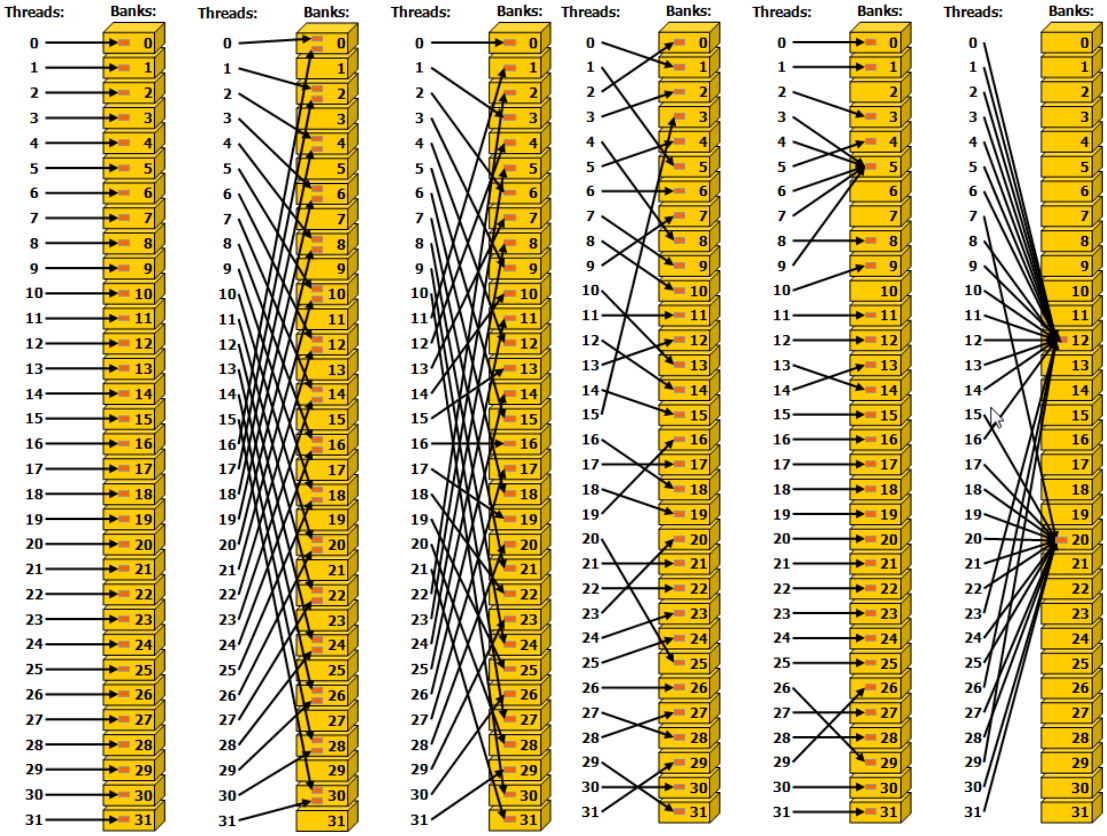


図 3.13 bank conflict が発生する・しない例 (4 byte 配列の場合)[1]

(bank 内側の橙色矩形はアドレスを指し、同一 bank に 2 つある場合はアドレスが異なる)

(i) ストライド幅 1：bank conflict なし

(ii) ストライド幅 2：bank conflict 発生 処理は 2 回に分けられる(2-way bank conflict)

(iii) ストライド幅 3：bank conflict なし

(iv) 不規則なアクセス：bank conflict なし

(v) thread 3, 4, 6, 7 が同じアドレス・bank にアクセス：bank conflict なし

(vi) 同じアドレス・bank にアクセスが集中：bank conflict なし

図 3.13 (i)及び(iii)では、bank conflict を発生させずに 128 byte のアクセスを一度に処理でき、最も効率が良い。(ii)では 128 byte のアクセスのために 2-way bank conflict が発生し、スループットは本来の$1/2$になる。(vi)では bank conflict は発生していないものの、一度の処理で 2 つの要素、すなわち$4 \times 2 = 8$ byte のデータにしかアクセスしないため、スループットは本来の$8/128 = 1/16$になる。

CUDA を利用したプログラミングにおいては、これらの特徴をもつ shared memory を利用して global memory へのアクセスを最小化することや、global memory へのストライドアクセスを避けることが重要となる。

c)　register, local memory

register は kernel 中の自動変数に対応する。register はオンチップメモリであり高帯域幅・低レイテンシだが、SM1 基、block、thread あたりの数に制限がある。kernel の要求に対して register が不足する場合、不足分には local memory が使用されるが、local memory はオフチップメモリであり、アクセスには global memory と同程度のコストが掛かる[6]。ゆえに、あまりにも多くの自動変数を必要とする kernel を記述すべきではない。

前項でも述べたように、block サイズや、block が要求する shared memory や register の量は、SM1 基に一度に駐在できる warp の数(theoretical occupancy)に影響を与える。Computation Capability 8.9 における、thread や各メモリ、occupancy に関連する仕様を以下に示す。ただし、表中の K は 1024 を表す。

表 3.1 Computation Capability 8.9 の仕様([1]より一部抜粋)

項目	値
Maximum x -dimension of a grid of thread blocks	2^{31}-1
Maximum y- or z-dimension of a grid of thread blocks	65535
Maximum dimensionality of thread block	3
Maximum x- or y-dimensionality of a block	1024
Maximum z-dimension of a block	64
Maximum number of threads per block	1024
Warp size	32
Maximum number of resident blocks per SM	24
Maximum number of resident warps per SM	48
Maximum number of resident threads per SM	1536
Number of 32-bit registers per SM	64 K
Maximum number of 32-bit registers per thread block	64 K
Maximum number of 32-bit registers per thread	255
Maximum amount of shared memory per SM	100 KB
Maximum amount of shared memory per thread block	99 KB
Number of shared memory banks	32

3.3　GPGPU を用いた核計算コード開発における課題

まず、3.2 節で述べたことを踏まえて、CUDA プログラミングにおいて注意すべき点を以下に挙げる[6]。

優先度 高:

- global memory アクセスを可能な限り coalesced access にする。
- global memory アクセスを可能な限り避け、shared memory が利用できる場合には優先的に利用する。
- warp 内の条件分岐を避ける。
- host・device 間のデータ転送を可能な限り減らす。

優先度 中:

- global memory への冗長なアクセスを shared memory の利用などにより減らす。
- レイテンシ隠蔽のため、streaming multiprocessor ごとに十分な数の active warp を維持する。すなわち、occupancy を高く維持する。
- block あたりの thread 数を 32 の倍数となるようにする。

ここで、CUDA を用いて核計算コードを開発する場合に課題となる点について以下で考察する。

まず、核計算コードにおいて一般に並列化可能なループ変数はエネルギー、空間メッシュのインデックス変数である。輸送計算コードの場合、飛行方向や角度中性子束の展開次数、characteristics line 等が加わることがある。CPU を 1 つ用いて計算する場合は、一般に十数の thread を発行すれば CPU の性能を十分に活用できる。一方で GPU を利用する場合、性能を十分に活かすには streaming multiprocessor を埋め尽くすための数百の block、すなわち数千～数万 thread が最低限必要となる。ゆえに、CPU においてはいずれか 1 つのループ変数について並列化すればよいのに対して、GPU においては複数のループ変数について並列化する必要があることが多い。空間メッシュ数が極端に多ければ、空間メッシュについて並列化するだけでよい場合もある。しかし、本研究で利用するS_N法のように空間メッシュについての計算どうしに依存性があり、単純に全空間メッシュについて並列化することができないこともある。

どのループ変数について並列化し、データをどのように格納し、block や grid をどのように構成するかは、thread どうしの処理の依存性やメモリアクセスの効率を踏まえて考える必要がある。

例えば本節冒頭で高優先度の目標として挙げた coalesced access を実現するには、連続し

た thread が連続したアドレスにアクセスすればよい。しかし、輸送計算においては各空間メッシュに対するアクセスの順序が中性子飛行方向によって異なる場合が多いため、空間メッシュインデックスi, j, kについて連続に格納されたデータに対して coalesced access を実現することは困難である。

ここで、各データをエネルギー群インデックスgについて連続となるよう格納する場合を考える。このとき、thread もエネルギー群について連続となるように構成すればデータに対する coalesced access を容易に実現できる。核計算における多くのデータはエネルギー群に依存するため、これは核計算コードにおいて coalesced access を実現するのに適した方法といえる。

一方で、thread どうしに依存性がある処理をしたい場合がある。例えば、付近の空間メッシュどうしでデータをやり取りしたい場合や、各 thread が計算した値の総和をとりたい場合などである。このような場合、global memory を介するのは非効率的であるため、可能な限り shared memory などの高速な手段[3]を利用する必要がある。ゆえに、データをやり取りしたい thread どうしはなるべく block 内に集まっていることが望ましい。しかし、block あたりの thread 数や shared memory の容量には制限があるため、注意が必要である。

以上のように、CUDA を用いたコードを開発する上でパフォーマンスに影響する要素は多数存在するため、開発前に各要素の影響を完全に予想し、すぐさま最適なコードを開発することは困難である。また、十分な並列化度・coalesced access・thread 間の効率的なデータのやり取りといった複数の目標を競合なく同時に達成することも困難を伴う。ゆえに、開発したコードに対してプロファイリングを実施することでパフォーマンスに影響を及ぼしている個所を特定し、改善していくことが求められる。本研究では、プロファイリングツールとして主に NVIDIA Nsight Compute[10]を利用する。

[3] thread どうしでデータを高速にやり取りする他の手段として、warp shuffle 関数が存在する。これら関数は warp 内 thread に限りデータを交換でき、shared memory を介さないためより高速な実行が期待できる。ただし、本研究では取り扱わない。

3.4 GPGPU を用いたS_N法に基づくα固有値計算の実装

本節では GPGPU を用いたS_N法に基づくα固有値計算の実装について述べる。

3.4.1 項では、S_Nに基づくα固有値計算の計算フローを示す。3.4.2 項及び 3.4.3 項では、S_N法の非等方散乱中性子源更新計算及び transport sweep について、それぞれ GPGPU を用いて並列化する手法について述べる。3.4.4 項では、計算に必要な global memory 及び shared memory の量を概算する。

本節では、第 2 章に引き続き以下に示す文字を使用する。ただし、C++に合わせて各インデックスが 0 から開始することに注意されたい。

i,j,k	：x,y,z方向空間メッシュインデックス
NX,NY,NZ	：x,y,z方向空間メッシュ分割数（$0 \le i < NX, 0 \le j < NY, 0 \le k < NZ$）
g	：エネルギー群インデックス
NG	：エネルギー群数（$0 \le g < NG$）
d	：中性子飛行方向
ND	：中性子飛行方向分割数（$0 \le d < ND$）
l,m	：実球面調和関数R_m^lに対応する展開次数
NL	：実球面調和関数展開次数の上限（$0 \le l \le NL,\ -l < m < l$）
$\phi_{m,g,i,j,k}^{l}$	：l,m,g,i,j,kに対応する中性子束モーメント
$\psi_{d,g,i,j,k}$	：d,g,i,j,kに対応する角度中性子束
$Q_{m,g,i,j,k}^{l}$	：l,m,g,i,j,kに対応する中性子源モーメント
$Q_{d,g,i,j,k}$	：d,g,i,j,kに対応する中性子源

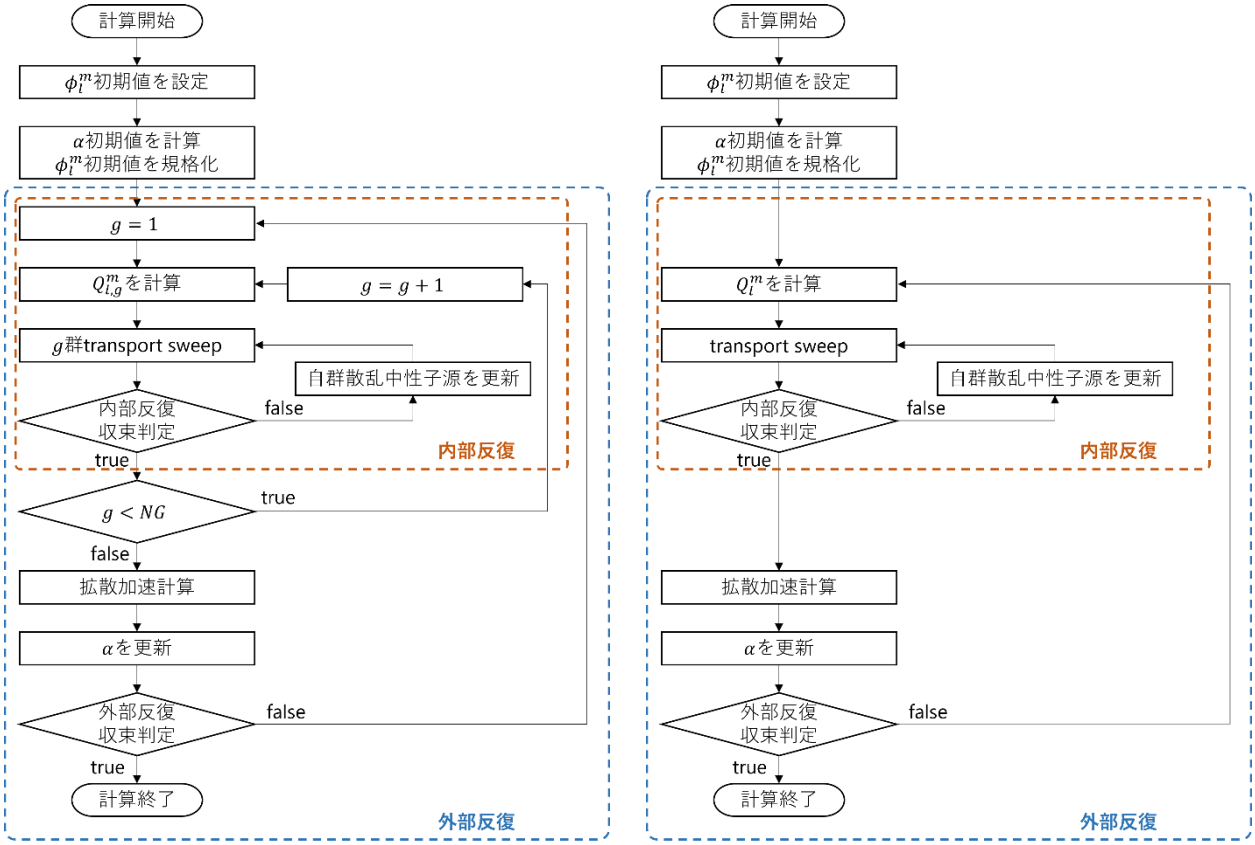
また、本節では変数g,i,j,kを省略する場合があることに注意されたい。省略した場合、特筆なき限り省略された添字の全範囲を表すものとする。例えば$Q_{g,d,i,j,k}$についてg,i,j,kを省略して単にQ_dとし、Q_dを計算すると記したときはg,i,j,kの全範囲について計算するものとする。ただし、実球面調和関数の展開係数l,m及び飛行方向dについては省略せず、Q_l^mやQ_dと記したときは特筆なき限りl,mやdの全範囲について計算するものとする。

3.4.1 計算フロー

第 2 章の理論に基づいた、S_N法に基づくα固有値計算の計算フローを図 3.14 に示す。

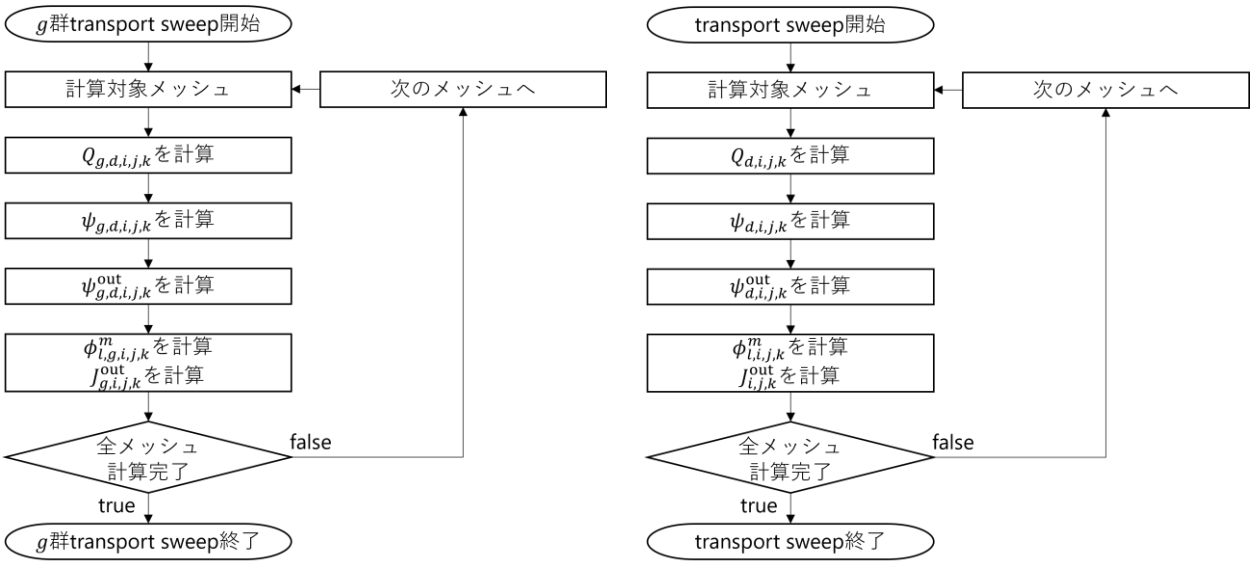
図 3.14 (i)は、n回目の外部反復におけるg群の散乱源$Q_{l,g}^{m(n)}$更新に、直前に更新された$g-1$群の中性子束$\phi_{l,g-1}^{m(n)}$を直ちに利用するガウスザイデル法を用いた場合のフローを示している。一方で(ii)ではn回目の外部反復における散乱源$Q_{l,g}^{m(n)}$更新に、前回の外部反復で更新された中性子束$\phi_l^{m(n-1)}$を利用するヤコビ法の場合を示している。ヤコビ法に比べ、ガウスザイデル法は収束性に優れるが、エネルギー群変数gについての計算を並列化することができない。よって本研究では、計算の並列化のためヤコビ法を採用した。

また、transport sweep の計算フローを図 3.15 に示す。(i), (ii)はそれぞれエネルギー群ごとに計算する場合と、全エネルギー群について同時に計算する場合のフローである。理由は後述するが、第 2 章の手順と異なり transport sweep にQ_l^mからQ_dを計算する手順が含まれることに注意されたい。



(i) 外部反復にガウスザイデル法を用いる場合　　(ii) 外部反復にヤコビ法を用いる場合

図 3.14 S_N法に基づくα固有値計算の計算フロー



(i) エネルギー群ごとに計算する場合　　(ii) 全エネルギー群について同時に計算する場合

図 3.15 transport sweep の計算フロー

本研究において GPGPU による並列化の対象とする計算は、特に計算量が多い非等方散乱中性子源Q_l^m更新と transport sweep、拡散加速計算の 3 つとする。

GPU による計算において性能を引き出すには、3.2 項で述べたように計算を十分多くの thread に並列化する必要がある。よって、並列化するループ変数はエネルギー群g、飛行方向d、空間メッシュインデックスi,j,kとする。ただし、transport sweep においては空間メッシュの計算同士に依存性があるため、全メッシュについて並列化することはできない。transport sweep の並列化手法については 3.4.3 項で述べるが、ここでは例として 16 メッシュについて並列化できるとする。このとき、例えば$NG = 172, ND = 72$、block サイズを 256 とすれば、並列化により発行される全 thread の数は$172 \times 72 \times 16 = 198{,}144$ thread、block 数は$198{,}144/256 = 774$となる。これは RTX 4090 の搭載 SM 基数 128 を超えており、搭載された SM を使い切るのに十分な block 及び warp を発行できているといえる。なお、実球面調和関数展開係数l,mについては並列化の対象外とする。

CPU 計算においては原則としてgのみを並列化対象のループ変数とする。ただし、使用するライブラリにより一部計算が並列化やベクトル化される場合がある。

ここで計算フローに関して、第 2 章で述べたものとは異なり、Q_l^mからQ_dを計算する手順を transport sweep のフローに含んでいることに注意されたい。これは、並列化に伴うメモリ消費量を抑えるためである。

非等方散乱中性子源更新と transport sweep をそれぞれ 1 つの kernel として、Q_l^mからQ_dを計算する手順を非等方散乱中性子源更新に含める場合を考える。CUDA において、kernel 終了時に global memory 以外にあるデータは失われるため、kernel 同士のデータの受け渡しの

ために$Q_{g,d,i,j,k}$の全要素を global memory に保持する必要がある。$Q_{g,d,i,j,k}$の全要素数は$NG \times ND \times NX \times NY \times NZ$であり、例えば$NG = 172, ND = 72, NX = NY = NZ = 32$、単精度(1 要素 4 byte, `float` 型)のとき、$4\ \text{byte} \times 172 \times 72 \times 32^3 = 1{,}623{,}195{,}648\ \text{byte} \approx 1.5\ \text{GB}$の global memory が必要となる。倍精度では、さらに 2 倍となる。これは飛行方向分割数NDに比例して増加するため、NDをさらに増加させると host memory に比べ容量の小さい global memory を大きく圧迫することになる。

一方で、展開係数で表された形$Q^m_{l,g,i,j,k}$の全要素数は、$NG \times (NL+1)^2 \times NX \times NY \times NZ$と表される。展開次数上限を大きめにとって$NL = 7$としても、必要なメモリは$4\ \text{byte} \times 172 \times (7+1)^2 \times 32^3 \approx 1.3\ \text{GB}$である。これは角度中性子束$\psi_d$と中性子束モーメント$\phi^m_l$の関係についても同様であり、計算全体にわたって保持するデータをなるべく展開係数で表された形ϕ^m_l, Q^m_lに限定することで、メモリ消費量を節約することができる。これを実現するため、Q^m_lからQ_dを計算する手順を transport sweep のフローに含め、非等方散乱中性子源更新と transport sweep をそれぞれ 1 つの kernel として実装する。kernel 中でQ^m_lからQ_dを計算するとき、各 thread は自身が担当するインデックスg, d, i, j, kに該当する$Q_{g,d,i,j,k}$の値を register に保持するだけでよい。

3.4.2　非等方散乱中性子源更新計算の並列化

前項で述べたように、非等方散乱中性子源更新計算では式(2.45)に基づき、非等方散乱断面積Σ_{sl}と中性子束モーメントϕ_l^mから非等方散乱中性子源モーメントQ_l^mを計算する。このときQ_dは計算せず、transport sweep 時に計算する。

$$Q_{l,g,i,j,k}^{m} = \sum_{g'=1}^{NG} \Sigma_{\mathrm{sl},g'\to g,i,j,k}\, \phi_{l,g',i,j,k}^{m} \tag{2.45}$$

再掲

CUDA を用いた非等方散乱中性子源更新計算実装のイメージを図 3.16 に示す。

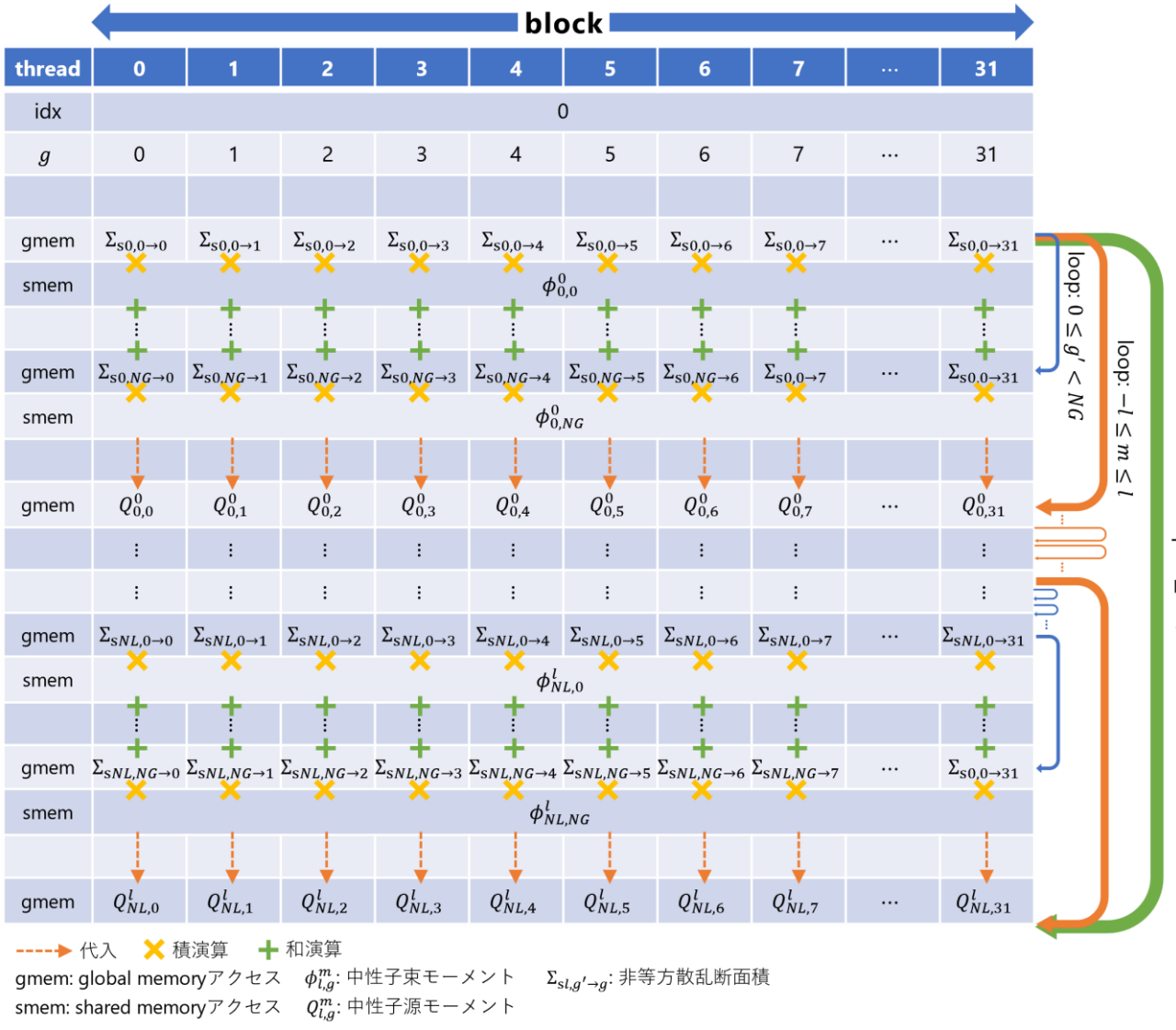


図 3.16 CUDA を用いた非等方散乱中性子源更新計算実装のイメージ

$(idx = 0, 0 \le g < 32)$

このコードでは、変数g, i, j, kについて並列化し、全空間メッシュの内の 1 メッシュ・全エネ

ルギー群の内 32 群を 1 つの block として処理する(このとき 1 block = 1 warp)。上図はその 1 つ目の block ($idx = 0, 0 \leq g < 32$)の処理を図示したものである。ただし、idxは$idx = NXNY \times k + NX \times j + i$で表される空間メッシュのインデックスである。このとき発行される block 数は、$NXNYNZ \times \mathrm{ceil}(NG, 32)/32$である。ここで、$\mathrm{ceil}(a, b)$は$a$を$b$の倍数に切り上げる関数であるとする。例えば、$NG = 172$のとき$\mathrm{ceil}(NG, 32)/32 = \mathrm{ceil}(172, 32)/32 = 6$となる。

エネルギー群数NGが 32 の倍数でない場合、thread に端数が生じるが、余計に発行した thread を休眠させることで対応する。NGが小さい場合休眠 thread が相対的に多くなり性能に問題が生じる可能性が考えられるが、この場合 block の構成を変更することで対応できると考えられる。例えば、block に 2 つ以上のメッシュを割り当て、エネルギー群の割当を減らすなどの対応が考えられる。

図 3.16 のように、コードは変数g, m, lについての 3 重ループで構成され、各回のループで中性子束モーメント$\phi_{l,g}^{m}$が全 thread にブロードキャストされる。このとき、global memory に格納された$\phi_{l,g}^{m}$へ直接アクセスすると、1 命令につき 1 要素にのみアクセスすることとなり、3.2 節で述べたようにこれは非効率的で性能劣化の原因となり得る。これを避けるため、本コードでは事前に一定範囲の$\phi_{l,g}^{m}$を shared memory にロードしておくことで、global memory へのアクセス回数を削減する。本コードでは 1 block = 32 thread であることから、$\phi_{l,g}^{m}$を 32 要素ずつ global memory から shared memory にロードする。32 要素を使い果たしたら、また新たに次の 32 要素をロードすることを繰り返す。これにより、global memory へアクセスする際は連続する 32 要素(128 byte $= 4 \times 32$ byte トランザクション、倍精度の場合はこれの 2 倍)へ一度にアクセスする coalesced access となり、効率的である。

他の変数$Q_{l}^{m}, \Sigma_{sl,g' \to g}$については global memory へのアクセスである。これらのデータがgについて連続となるように格納されていれば、図 3.16 からわかるように毎回のアクセスは連続する 32 要素へ一度にアクセスする coalesced access となる。

$\Sigma_{sl,g' \to g}$はmについてのループで再利用されるが、$32 \times NG$要素分の$\Sigma_{sl,g' \to g}$を shared memory に格納することは容量の観点から難しいため、shared memory は利用していない。mのループを最内周にすることで可能となるが、上述した$\phi_{l,g}^{m}$の coalesced access 化と競合するため、見送ることとした。shared memory は利用していないものの、他の block でも$\Sigma_{sl,g' \to g}$の同じ要素にアクセスすることは多く起こりうるため、L2 キャッシュにヒットする可能性は高く、ある程度のスループットが期待できると考えられる。

また、この計算は大量の行列・ベクトル積計算の集合とみなすことができ、cuBLAS[13]の`cublasSgemvBatched()`関数などを利用することもできる。ただし、一度の呼出で計算する範囲のメッシュが均質である(参照する断面積が同じ)ことが条件となる。断面積が異なる計

算をするには複数回 `cublasSgemvBatched()`を呼び出す必要がある。stream[4]を利用することで cuBLAS 関数を同時に複数実行することは可能であるが、同時実行できる数には制限があることと、多数回の呼び出しはオーバーヘッドを伴うことに注意しなければならない。

3.4.3 transport sweep の並列化

transport sweep では、与えられた中性子源Q_l^mに対して、全空間メッシュ・各飛行方向について式(2.33)及び式(2.21)に基づき平均角度中性子束ψを更新し、その値を用いて式(2.22)、式(2.34)、式(2.56)–(2.61)に基づき中性子モーメントϕ_l^m及び中性子流J_x, J_y, J_zを計算する。

$$Q_{g,d,i,j,k} = \sum_{l=0}^{NL} \frac{2l+1}{4\pi} \sum_{m=-l}^{l} Q_{l,g,i,j,k}^{m} R_l^m(\mu_d, \eta_d, \xi_d) \tag{2.33}$$
再掲

$$\psi_{g,d,i,j,k} = \frac{Q_{g,d,i,j,k} + 2\left(\frac{|\mu_d|}{\Delta x_{\mathrm{i}}}\psi_{g,d,i,j,k}^{xin} + \frac{|\eta_d|}{\Delta y_j}\psi_{g,d,i,j,k}^{yin} + \frac{|\xi_d|}{\Delta z_k}\psi_{g,d,i,j,k}^{zin}\right)}{\left(\Sigma_{\mathrm{t},g,i,j,k} - \frac{\alpha}{v_g}\right) + 2\left(\frac{|\mu_d|}{\Delta x_{\mathrm{i}}} + \frac{|\eta_d|}{\Delta y_j} + \frac{|\xi_d|}{\Delta z_k}\right)} \tag{2.21}$$
再掲

$$\begin{aligned} \psi_{g,d,i,j,k}^{xout} &= 2\psi_{g,d,i,j,k} - \psi_{g,d,i,j,k}^{xin} \\ \psi_{g,d,i,j,k}^{yout} &= 2\psi_{g,d,i,j,k} - \psi_{g,d,i,j,k}^{yin} \\ \psi_{g,d,i,j,k}^{zout} &= 2\psi_{g,d,i,j,k} - \psi_{g,d,i,j,k}^{zin} \end{aligned} \tag{2.22}$$
再掲

$$\phi_{l,g,i,j,k}^{m} = \sum_{d=1}^{ND} w_d \psi_{g,d,i,j,k} R_l^m(\mu_d, \eta_d, \xi_d) \tag{2.34}$$
再掲

$$J_{g,i,j,k}^{x-} = 4\pi \sum_{d=1}^{ND} w_d \psi_{g,d,i,j,k}^{x-} \mu_d \tag{2.56}$$
再掲

$$J_{g,i,j,k}^{x+} = 4\pi \sum_{d=1}^{ND} w_d \psi_{g,d,i,j,k}^{x+} \mu_d \tag{2.57}$$
再掲

$$J_{g,i,j,k}^{y-} = 4\pi \sum_{d=1}^{ND} w_d \psi_{g,d,i,j,k}^{y-} \eta_d \tag{2.58}$$
再掲

$$J_{g,i,j,k}^{y+} = 4\pi \sum_{d=1}^{ND} w_d \psi_{g,d,i,j,k}^{y+} \eta_d \tag{2.59}$$
再掲

$$J_{g,i,j,k}^{z-} = 4\pi \sum_{d=1}^{ND} w_d \psi_{g,d,i,j,k}^{z-} \xi_d \tag{2.60}$$
再掲

$$J_{g,i,j,k}^{z+} = 4\pi \sum_{d=1}^{ND} w_d \psi_{g,d,i,j,k}^{z+} \xi_d \tag{2.61}$$
再掲

[4] stream: CUDA において GPU 上の処理を管理するキュー(queue)のこと。複数の stream を使用することで、複数の kernel やメモリ転送を同時実行することができる。

ただし、2.2.9 項で述べたようにある空間メッシュの計算は流入元メッシュの計算に依存するため、全空間メッシュについて並列化することはできない。変数g,dについてのみ並列化する場合、全 thread 数は$NG \times ND$であり、$NG = 172, ND = 72$のとき$172 \times 72 = 12{,}384$ thread となる。これは全 SM が同時に命令を発行できる thread 数を下回っており、GPU の性能を活用するには thread 数が不足しているといえる。また、後述するように処理を飛行方向の象限ごとに分割する場合、さらに不足する。

そこで、本研究では tiled hyperplane transport sweep[11]と呼ばれる手法を利用する。

tiled hyperplane transport sweep について説明する前に、まず hyperplane transport sweep(あるいは wavefront sweep)[12]について説明する。$\mu > 0, \eta > 0, \xi > 0$のとき、$i + j + k = 3h$ (hは整数)を満たすメッシュの組を hyperplane とする(図 3.17)。$i + j + k = 3(h-1)$の hyperplane(hyperplane hにとって流入元の hyperplane)に属するメッシュの計算が完了していれば、$i + j + k = 3h$の hyperplane に属するメッシュの計算は独立して、並列に計算することができる。これを利用することで、$NG \times ND \times$(hyperplane に属するメッシュ数) thread を発行できる。しかし hyperplane sweep では、hyperplane に属するメッシュ数が hyperplane の進行(hの増加)とともに変化する。ゆえに、hごとに thread を再発行する(kernel を呼び出す)か、十分な数の thread を発行したうえでその内いくつかもしくは大半を休眠させる必要がある。しかし kernel 呼び出しはオーバーヘッドを伴うため、短時間に複数回呼び出すことは現実的ではない。また、発行した thread の多くを休眠させることもリソースの無駄となる。

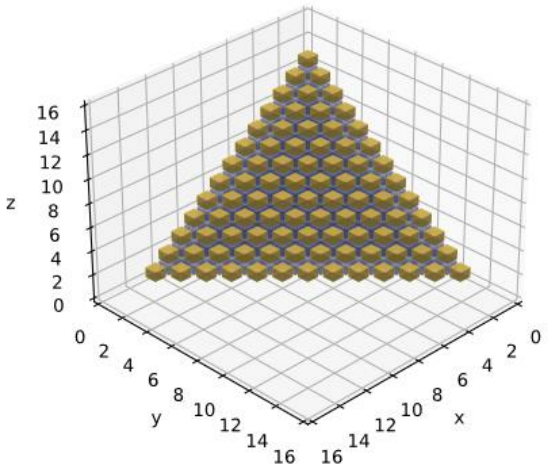


図 3.17 hyperplane transport sweep における hyperplane

そこで、tiled hyperplane transport sweep を利用する。本手法では、例えば$\mu > 0, \eta > 0, \xi > 0$のとき$yz$平面側から見た体系をいくつかの領域に分割し、その中で hyperplane を構築する(図 3.18)。これにより、sweep 開始地点と終了地点を除いて hyperplane の進行中も hyperplane に属するメッシュ数が変化せず、前述した hyperplane transport sweep の問題を解決できる。また、hyperplane 進行中に各 thread が担当するメッシュの(j, k)の組をしばらく一定に保つことができ、hyperplane 同士のデータのやり取りが簡単になる[11]。hyperplane に属するメッシュ数をNM_hで一定とすると、発行できる thread 数は$NG \times ND \times NM_h$ thread と表すことができ、例えば$NG = 172, ND = 72, NM_h = 16$であれば、$172 \times 72 \times 16 = 198{,}144$ thread となる。これは GPU の計算能力を使い切るのに十分な数であるといえる。また、飛行方向余弦μ, η, ξの正負の各組についても、同様のことがいえる。

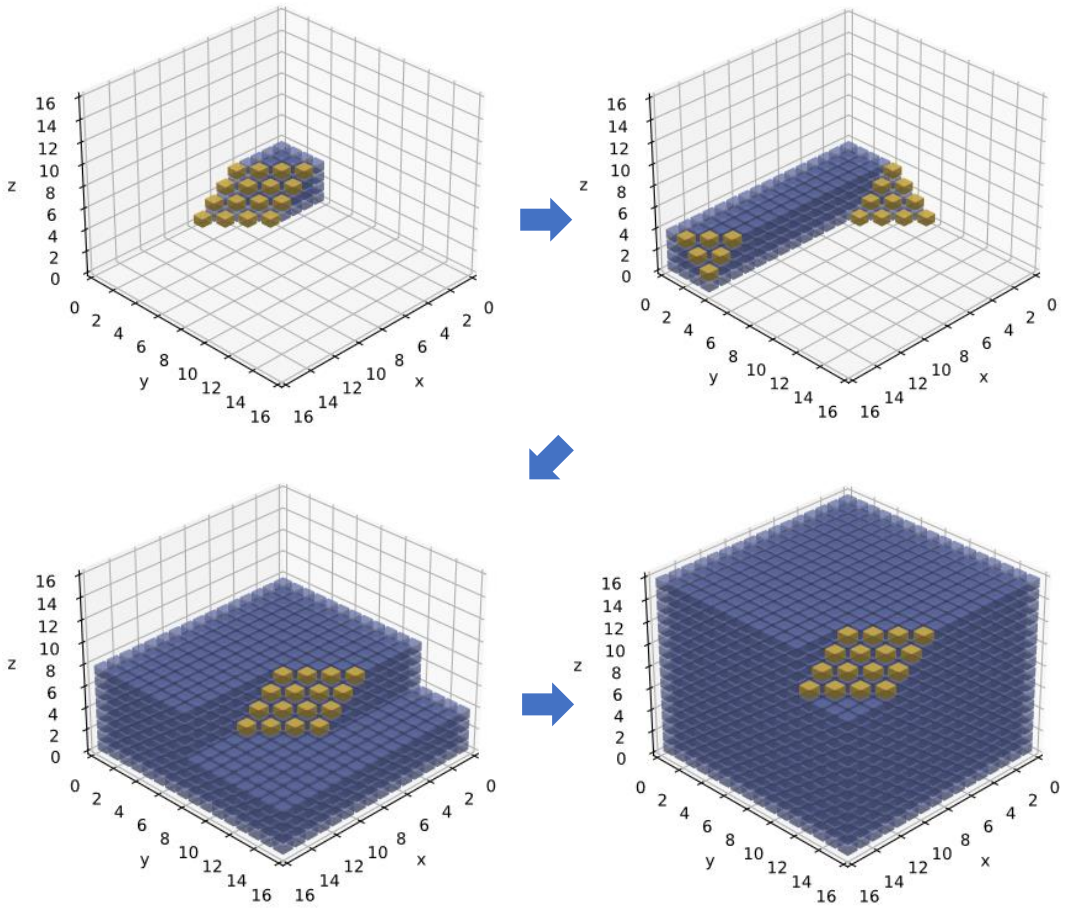


図 3.18 tiled hyperplane transport sweep ($\mu > 0, \eta > 0, \xi > 0$、hyperplane サイズ 16 の場合)
(黄: 計算対象 hyperplane、青: 計算終了)

ここで、tiled hyperplane transport sweep の計算フローと計算中の各変数の生存期間及び依存関係を下図に示す。図に示したデータの依存関係から、Q_d, ψ_dについてはある hyperplane についての計算中のみ生存していればよいことが分かる。よって、これらのデータは global memory へ格納する必要がなく、各 thread の register でデータを維持することができる。$\psi_d^{\mathrm{in/out}}$については、前後の hyperplane 及び hyperplane 境界部と流入/流出角度中性子束$\psi_d^{\mathrm{in/out}}$の値をやりとりする必要がある。

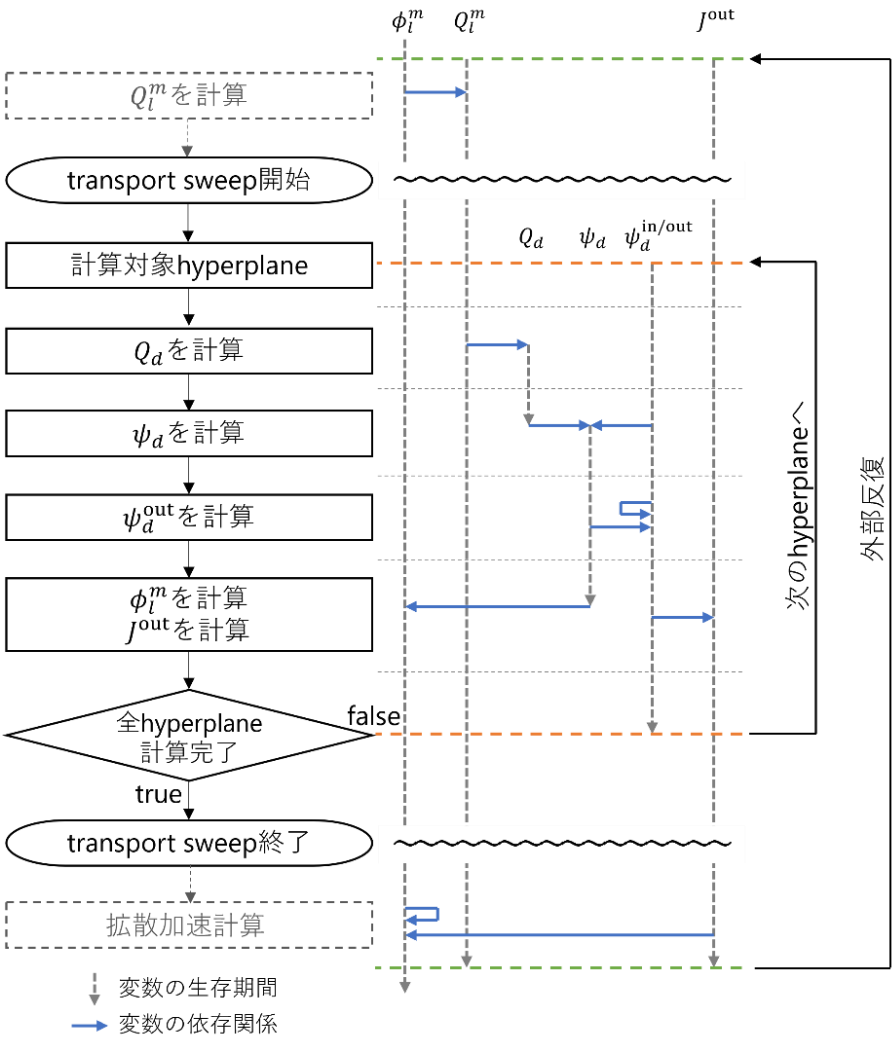
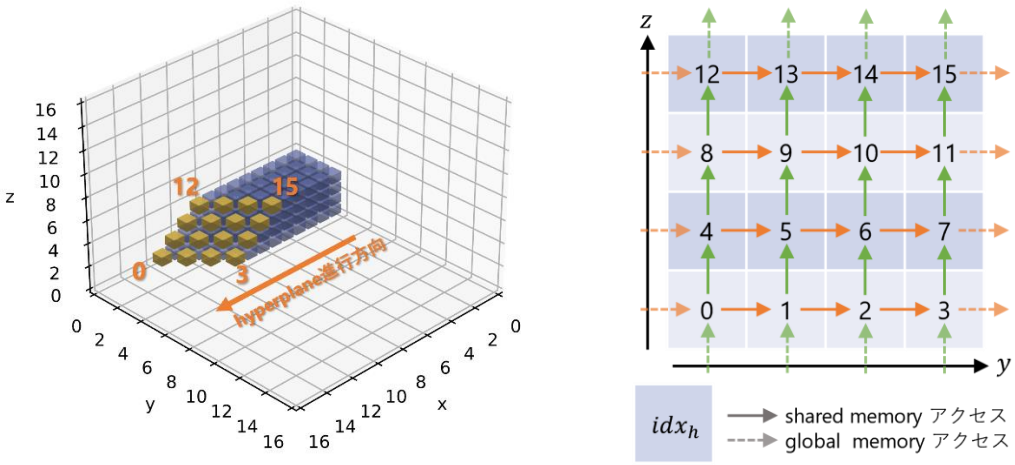


図 3.19 tiled hyperplane transport sweep における各変数の生存期間と依存関係
(元の変数に戻る矢印で表された依存関係は、自身の更新に自らの値を使うことを表す)

ここで、エネルギー群と飛行方向の一部をNG_h $(NG_h \leq NG)$群、ND_h $(ND_h \leq ND)$方向として、NM_h個の空間メッシュ、エネルギーNG_h群、飛行方向ND_h方向を 1 block に割り当てることを考える。このとき、1 block に対応する thread 数は、$NM_h \times NG_h \times ND_h$ threadとなる。

そして参考文献[11]の手法に則り、前後の hyperplane 同士のやり取りには shared memory を利用し、体系境界部及び hyperplane 境界部とのやり取りには global memory を利用する。また、流出/流入角度中性子束用の global memory については、体系境界および hyperplane 境界の分だけを確保すればよく、全領域分の global memory を確保する必要はない。

ここで、hyperplane に属する空間メッシュについて、下図(i),(ii)のようにサブインデックスidx_hを割り当てる。このとき、hyperplane における global / shared memory とのやり取りは下図(ii)のようになる。



(i) hyperplane とメッシュインデックス　　(ii) hyperplane 内のデータフロー

図 3.20 tiled hyperplane transport sweep における hyperplane 内のメッシュインデックスと hyperplane 内のデータフロー

例えば、$NM_h = 16, NG_h = 2, ND_h = 2$のとき、図 3.20(ii)に対応した block 内 thread 同士のデータのやり取りは、図 3.21 のように表すことができる。図に示された block は `blockIdx.x=blockIdx.y=blockIdx.z=0` の block である。この例では、threadIdx.x にdを、threadIdx.y にgを、threadIdx.z にidxを割り当てることを想定している。図から、hyperplane 境界を除き、block 内で流出/流入角度中性子束データのやり取りが可能であることを確認できる。

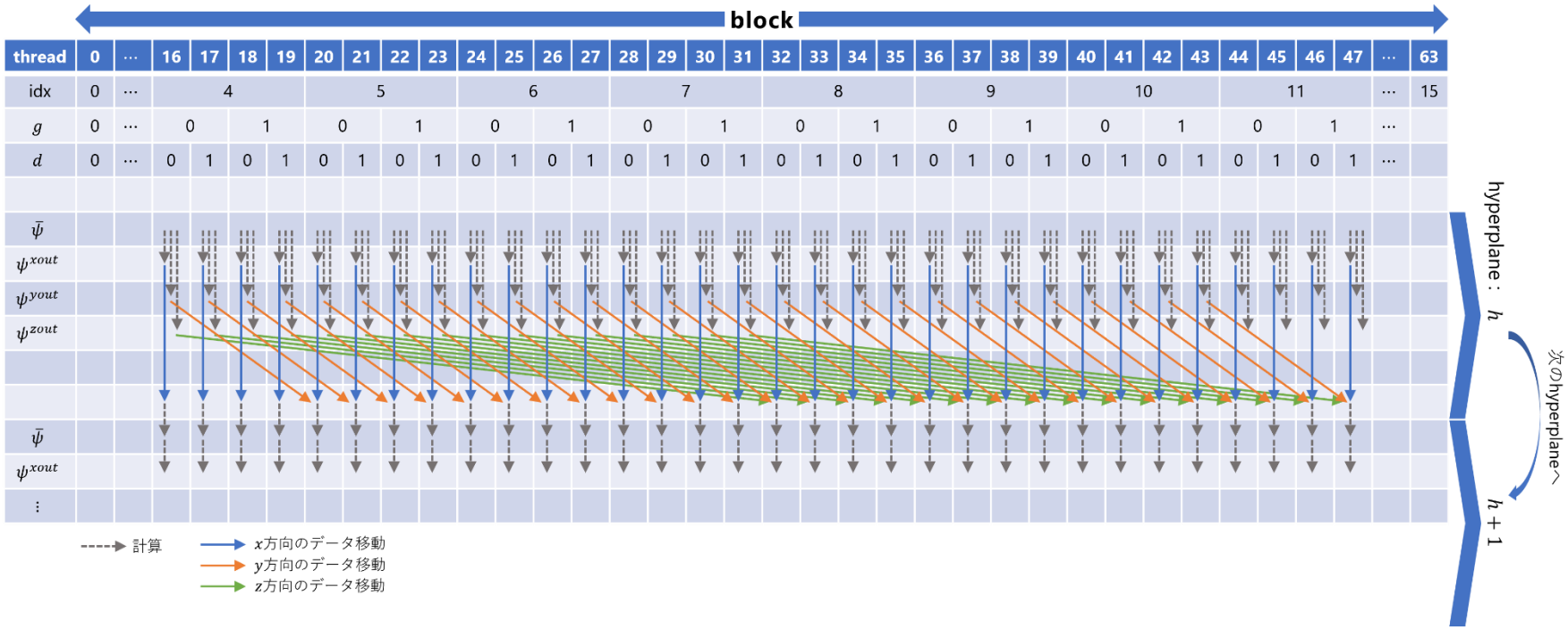


図 3.21 tiled hyperplane transport sweep における block 構造と block 内 thread 間のデータ依存関係

また、$NG = 172, ND = 72$、図 3.21 の場合における block と grid 割り当ての例を示す。発行される block 数は$\text{ceil}(NG, NG_h)/NG \times \text{ceil}(ND, ND_h)/ND$となる。ただし、kernel の実装を飛行方向象限ごとに分ける場合、NDは各象限に属する飛行方向の数となる。

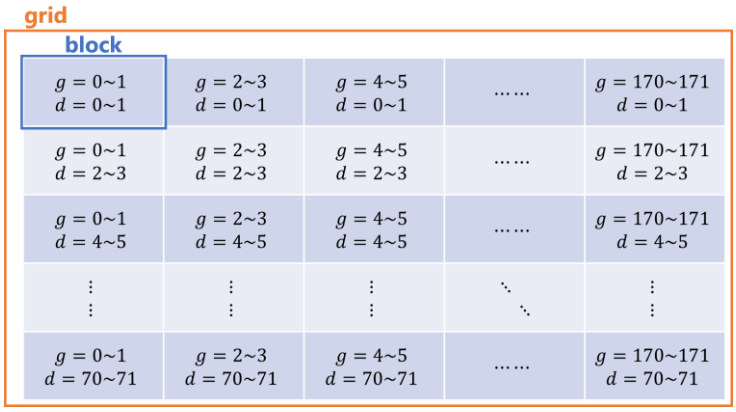


図 3.22 tiled hyperplane transport sweep における block, grid 割り当ての例

ここからは、tiled hyperplane transport sweep の実装のうち、特に中性子束モーメントϕ_l^m計算の実装について議論する。

ϕ_l^mの計算では、式(2.34)に基づき展開次数(l, m)の各組について、$w_d \psi_{g,d,i,j,k} R_l^m(\mu_d, \eta_d, \xi_d)$の全飛行方向の総和を計算する必要がある。各 thread は自らが担当するエネルギー、飛行方向、メッシュについて$\psi_{g,d,i,j,k}$を計算したうえで、展開次数(l, m)の全組についてループし$w_d \psi_{g,d,i,j,k} R_l^m(\mu_d, \eta_d, \xi_d)$を計算、対応する$\phi_{l,g,i,j,k}^m$のメモリへ加算する必要がある。このとき、発行された thread の中には同じg, i, j, k、異なるdを担当する thread が複数存在するため、加算操作において(g, i, j, k)の組が一致した各 thread が同一アドレスに同時にアクセスし、操作が競合する可能性がある。CUDA において複数 thread により全く同時に加算操作が行われた場合、どのような動作が発生するかは未定義であり、正常な計算結果を得られない。これは block 内 thread 同士であっても、block 同士であっても同様である。

よって、競合を解決するためにこの加算操作は不可分操作(あるいはアトミック操作、atomic operation)でなくてはならず、不可分操作に対応した実装が必要となる。CUDA においては`atomicAdd()`関数などの不可分操作に対応した関数が用意されており、global / shared memory 対する不可分操作及び、float, double 等の変数型に対応している。

しかし、`atomicAdd()`関数ではある thread によるロード、加算、格納の一連の操作が終了するまで、他の thread による同一アドレスへの操作を禁止する(ロックする)。ゆえに、`atomicAdd()`は実行コストが高く、多数回の呼び出しはボトルネックとなり得る。展開次数(l, m)の組み合わせの数は$(NL + 1)^2$で表されるため、展開次数上限NLの増加に伴い累乗でϕ_l^mの計算コストが上昇する。また、当然ながら global memory への不可分操作よりも、shared memory に対する不可分操作の方がコストは小さい。

よって、`atomicAdd()`の呼び出しを減らし、特に global memory に対する `atomicAdd()`を

いかに回避するかが求められる。以下では、global memory に対する `atomicAdd()`の呼び出しをどのようにして減らす実装をするかについて議論する。まず、簡易的な実装(ベースライン実装)である version 1 について、その次に `atomicAdd()`の呼び出しを減らすよう改良した version 2, 3 について述べる。

ここで各 version において共通の仕様について述べる。まず、`threadIdx.x` に変数g, dのどちらを対応させるかで、G-major(または Gmaj), D-major(または Dmaj)と分ける。D-major では、`threadIdx.x` に変数dを対応させ、それに伴いg, dの両方をインデックスに持つデータはdについて連続となるように格納している。G-major についても同様である。また、global memory のϕ_m^lはgについて連続になるように格納する。

a) version 1

version 1 は最も簡易的な実装であり、全 thread が global memory に対する `atomicAdd()`を$(NL+1)^2$回呼び出す。この実装において、各 thread は以下のように動作する。

1. $\psi_{g,d,i,j,k} \times w_d \times R_l^m$を各展開次数$(l, m)$について計算する
2. 1.の結果を global memory の$\phi_{m,g,i,j,k}^l$に加算する (global memory に対する `atomicAdd`)
3. $lm = lm + 1$、$lm = (NL+1)^2$のとき 5.へ
4. 1.へ戻る
5. 次の hyperplane へ

$NG_h = 2, ND_h = 8, NM_h = 16, NL = 2$, D-major の場合における各 thread の動作イメージを図 3.23 に示す。図に示された block は `blockIdx.x=blockIdx.y=blockIdx.z=0` の block である。ただし、図中のlmは(l, m)の組に対して一意に対応するインデックスであり、$0 \leq lm < (NL+1)^2$を満たすとする。また、idxは$idx = NX \times NY \times k + NX \times j + i$で表される空間メッシュのインデックスである。2 つ以上の thread により同時にアクセスされる可能性のあるデータは不可分操作により操作する必要がある。

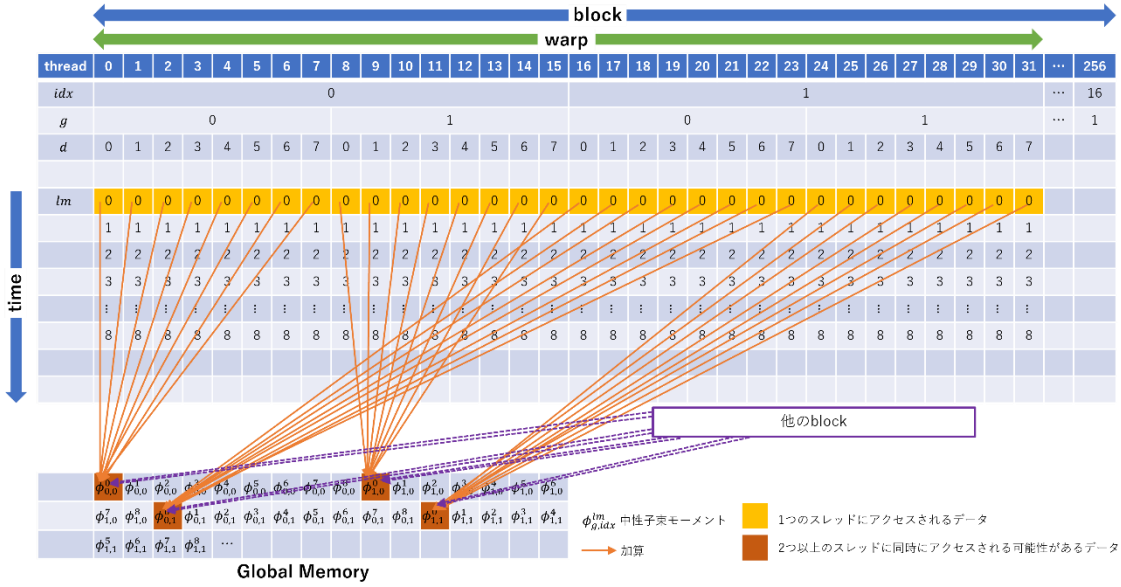


図 3.23 version 1 D-major ループ 0 回目($lm = 0$)の動作イメージ

ここで、$NX = NY = NZ = 32, NG = 172, ND = 1200$、全領域を均質とした体系をテスト体系とする。パラメータNM_h, NG_h, ND_hの組み合わせのいくつかで version 1 コードによりテスト体系について transport sweep を実施した。ただし解析対象は第 0 象限飛行方向について計算する kernel のみとする。計算時間の測定結果を図 3.24 に示す。ただし、高速化の見込みがないと判断したか、メモリなどの制限上実行できないパラメータについては解析を実施していない。また、そのうち 1 つの場合におけるコードの解析結果を図 3.25 に示す。

GmajV1

NMh=64	NGh					
	1	2	4	8	16
NDh 1					2314
2				1738	
4			1418		
8		1249			
16	1610				
32					

NMh=16	NGh					
	1	2	4	8	16
NDh 1					
2					1096
4				941.92	
8			893.93		
16		1020			
32	1261	978.42			

DmajV1

NMh=64	NGh					
	1	2	4	8	16
NDh 1					
2				822.25	
4			889.91		
8		1080			
16	1657				
32					

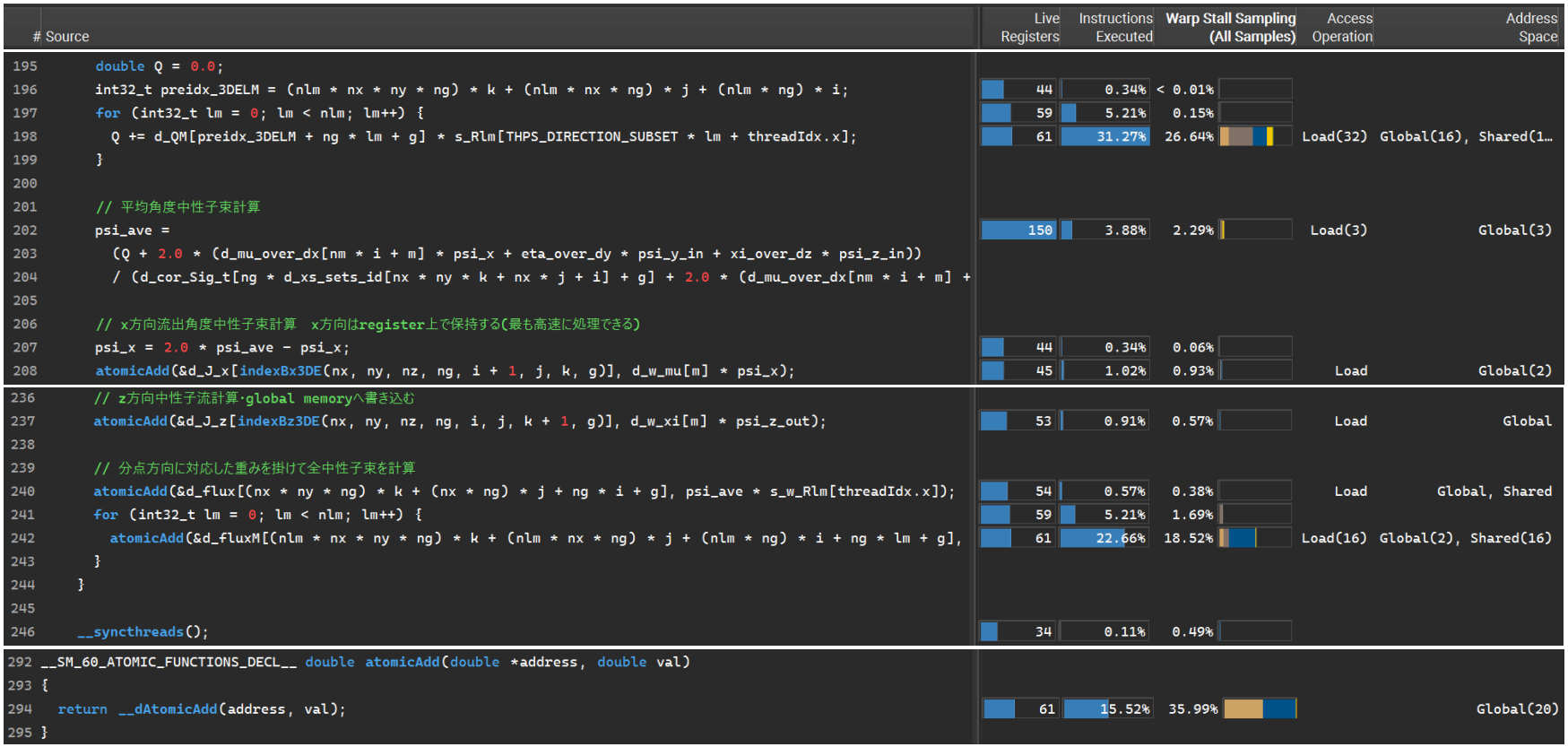
NMh=16	NGh					
	1	2	4	8	16
NDh 1					
2					
4					839.36
8				847.48	
16			982.49		
32	1299	1190			

図 3.24 tiled hyperplane transport sweep　version 1 計算時間 [ms]

$NX = NY = NZ = 32, NG = 172, ND = 1200, NL = 7$、第 0 象限のみ

NVIDIA Nsight Compute によるプロファイリング結果 (繰り返し回数: 5)

青←計算時間小　計算時間大→赤



#	Source	Live Registers	Instructions Executed	Warp Stall Sampling (All Samples)	Access Operation	Address Space
195	`double Q = 0.0;`					
196	`int32_t preidx_3DELM = (nlm * nx * ny * ng) * k + (nlm * nx * ng) * j + (nlm * ng) * i;`	44	0.34%	< 0.01%		
197	`for (int32_t lm = 0; lm < nlm; lm++) {`	59	5.21%	0.15%		
198	`Q += d_QM[preidx_3DELM + ng * lm + g] * s_Rlm[THPS_DIRECTION_SUBSET * lm + threadIdx.x];`	61	31.27%	26.64%	Load(32)	Global(16), Shared(1…
199	`}`					
200						
201	`// 平均角度中性子束計算`					
202	`psi_ave =`	150	3.88%	2.29%	Load(3)	Global(3)
203	`(Q + 2.0 * (d_mu_over_dx[nm * i + m] * psi_x + eta_over_dy * psi_y_in + xi_over_dz * psi_z_in))`					
204	`/ (d_cor_Sig_t[ng * d_xs_sets_id[nx * ny * k + nx * j + i] + g] + 2.0 * (d_mu_over_dx[nm * i + m] +`					
205						
206	`// x方向流出角度中性子束計算　x方向はregister上で保持する(最も高速に処理できる)`					
207	`psi_x = 2.0 * psi_ave - psi_x;`	44	0.34%	0.06%		
208	`atomicAdd(&d_J_x[indexBx3DE(nx, ny, nz, ng, i + 1, j, k, g)], d_w_mu[m] * psi_x);`	45	1.02%	0.93%	Load	Global(2)
236	`// z方向中性子流計算･global memoryへ書き込む`					
237	`atomicAdd(&d_J_z[indexBz3DE(nx, ny, nz, ng, i, j, k + 1, g)], d_w_xi[m] * psi_z_out);`	53	0.91%	0.57%	Load	Global
238						
239	`// 分点方向に対応した重みを掛けて全中性子束を計算`					
240	`atomicAdd(&d_flux[(nx * ny * ng) * k + (nx * ng) * j + ng * i + g], psi_ave * s_w_Rlm[threadIdx.x]);`	54	0.57%	0.38%	Load	Global, Shared
241	`for (int32_t lm = 0; lm < nlm; lm++) {`	59	5.21%	1.69%		
242	`atomicAdd(&d_fluxM[(nlm * nx * ny * ng) * k + (nlm * nx * ng) * j + (nlm * ng) * i + ng * lm + g],`	61	22.66%	18.52%	Load(16)	Global(2), Shared(16)
243	`}`					
244	`}`					
245						
246	`__syncthreads();`	34	0.11%	0.49%		
292	`__SM_60_ATOMIC_FUNCTIONS_DECL__ double atomicAdd(double *address, double val)`					
293	`{`					
294	`return __dAtomicAdd(address, val);`	61	15.52%	35.99%		Global(20)
295	`}`					

図 3.25 tiled hyperplane transport sweep　version 1　コード解析結果(一部抜粋)

$NX = NY = NZ = 32, NG = 172, ND = 1200, NL = 7$、第 0 象限のみ、$NM_h = 16, NG_h = 8, ND_h = 2$、D-major

NVIDIA Nsight Compute によるプロファイリング結果(繰り返し回数: 5)

図 3.24 より、パラメータ次第で最大 3 培程度性能が変化することがわかる。性能変化の要因はキャッシュヒット率やメモリアクセスの効率、各種命令の実行コストなど、多数の要因があると考えられ、特定は難しい。

特に性能劣化の要因を特定するため、図 3.25 に示したコード解析結果について考察する。図中の Instructions Executed はその行の命令実行数[%]、Warp Stall Sampling は stall に陥った warp がサンプリングされた数[%]を示している。Warp Stall Sampling の色は、どの要因で warp が stall に陥ったかにより異なり、薄茶は Lg Throttle、濃茶は Long Scoreboard、濃青は MIO Throttle、黄は Short Scoreboard を示している。詳しくは公式ドキュメントを参考にされたいが、おおむね、Lg Throttle 及び Long Scoreboard は global memory アクセスによるもの、MIO Throttle 及び Short Scoreboard は shared memory アクセスを含む L1 キャッシュアクセスによるものであると考えてよい。

242 行目は見切れているが、$\psi_{g,d,i,j,k} \times w_d \times R_l^m$を計算し `atomicAdd()`を呼び出している。ただし、242 行目の解析結果は `atomicAdd()`の実行を含んでおらず、`atomicAdd()`の解析結果は 294 行目に集約されていることに注意されたい。

解析結果より、展開次数lmのループ部分と、`atomicAdd()`に関連する部分で Warp Stall Sampling が極めて多いことがわかる。特に 242, 297 行目を合わせると Warp Stall Sampling の約 54.5%を占める。これはlmについてのループのため実行回数が多い(具体的には$(NL+1)^2 = 64$回)ことや、`atomicAdd()`の実行コストが高いことであると考えられる。

また、198 行目の Lg Throttle 及び Long Scoreboard の要因は global memory のQ_m^lへのアクセスの多さと、そのアクセスが同じgを担当する thread へのブロードキャストになっている、すなわち一度にアクセスするデータ量が 32 byte を下回っているためであると考えられる。また、MIO Throttle 及び Short Scoreboard については shared memory のR_m^lへのアクセスの多さが原因と考えられる。

version 2, 3 ではこれらの問題(の一部)について改善を図る。

b) version 2

global memory に対する atomicAdd が問題となるのであれば、block 内で一旦データを shared memory にまとめて、そののちに global memory へ加算するようにすれば、性能の改善が可能ではないかと考え、version 2 を実装した。

この場合、各スレッドは以下のように動作する。

1. $\psi_{g,d,i,j,k} \times w_d \times R_l^m$を各展開次数$(l,m)$について計算する
2. 1.の結果を shared memory の$\phi_{m,g,i,j,k}^l$に加算する (shared memory に対する `atomicAdd`)
3. $lm = lm + 1$、$lm = (NL+1)^2$のとき 5.へ
4. 1.へ戻る
5. block 内のスレッドを同期 (block 内の shared memory に対する操作の完了を待つ)
6. $lm = 0$
7. (同じgを担当するスレッドのうち 1 スレッドのみ動作 他のスレッドは休眠)
   shared memory の$\phi_{m,g,i,j,k}^l$を global memory の$\phi_{m,g,i,j,k}^l$へ加算する(global memory に対する `atomicAdd`)
8. $lm = lm + 1$、$lm = (NL+1)^2$のとき 9.へ
9. 次の hyperplane へ

これにより、global memory に対する `atomicAdd()`の呼び出し回数を$1/ND_h$に減らすことができる。global, shared memory の同一アドレスにアクセスすることは避けられていないため、global, shared memory に対する不可分操作は依然として必要である。また、`atomicAdd()`の呼び出しの合計回数は、shared と global で段階を踏んでいる分増加する。

$G_h = 2, ND_h = 8, NM_h = 16, NL = 2$の場合における各 thread の動作イメージを図 3.26 に示す。図に示された block は `blockIdx.x=blockIdx.y=blockIdx.z=0` の block である。

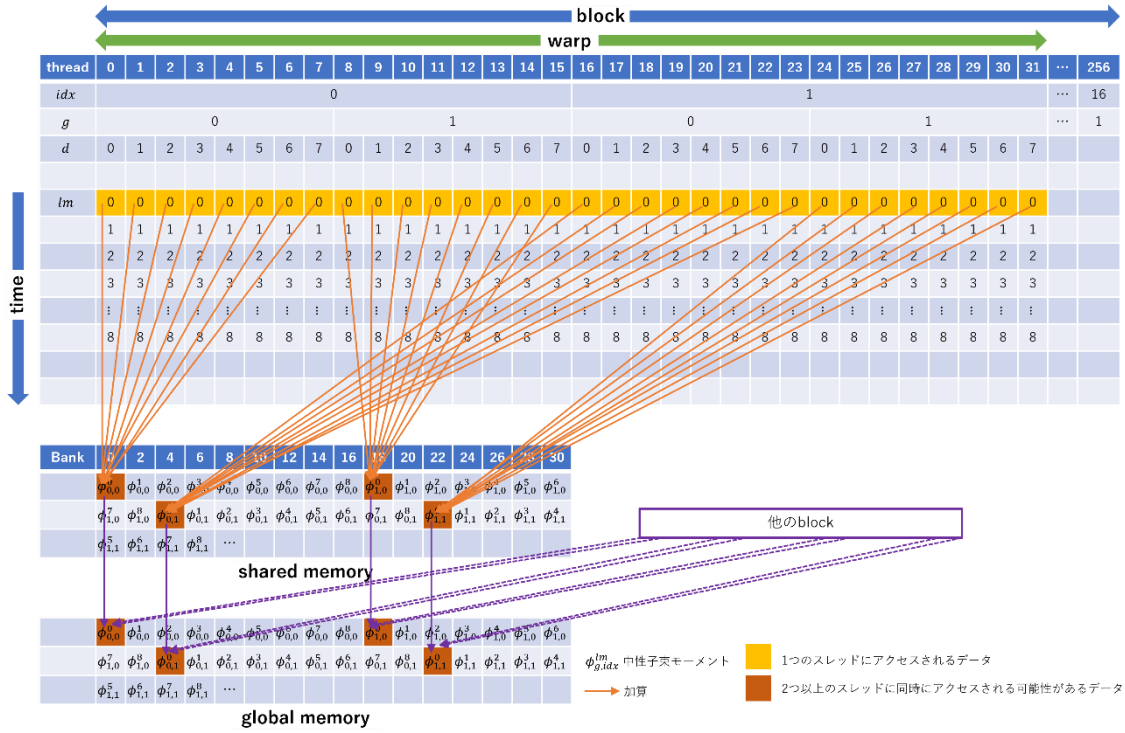


図 3.26 version 2 ループ 0 回目($lm = 0$)の動作イメージ
(global/shared memory アクセスは別個のループで発生することに注意)

ここで、version 1 と同様に、パラメータNM_h, NG_h, ND_hの組み合わせのいくつかで version 2 コードによりテスト体系について transport sweep を実施した。ただし解析対象は第 0 象限飛行方向について計算する kernel のみとする。計算時間の測定結果を図 3.27 に示す。また、そのうち 1 つの場合におけるコードの解析結果を図 3.28 に示す。

GmajV2

NMh=64	NGh					
	1	2	4	8	16
NDh 1						
2						
4						
8		1510				
16	2380					
32						

NMh=16	NGh					
	1	2	4	8	16
NDh 1						
2						
4				1071		
8			1316			
16		2289				
32	3677	2518				

DmajV2

NMh=64	NGh					
	1	2	4	8	16
NDh 1						
2						
4						
8		1301				
16	2336					
32						

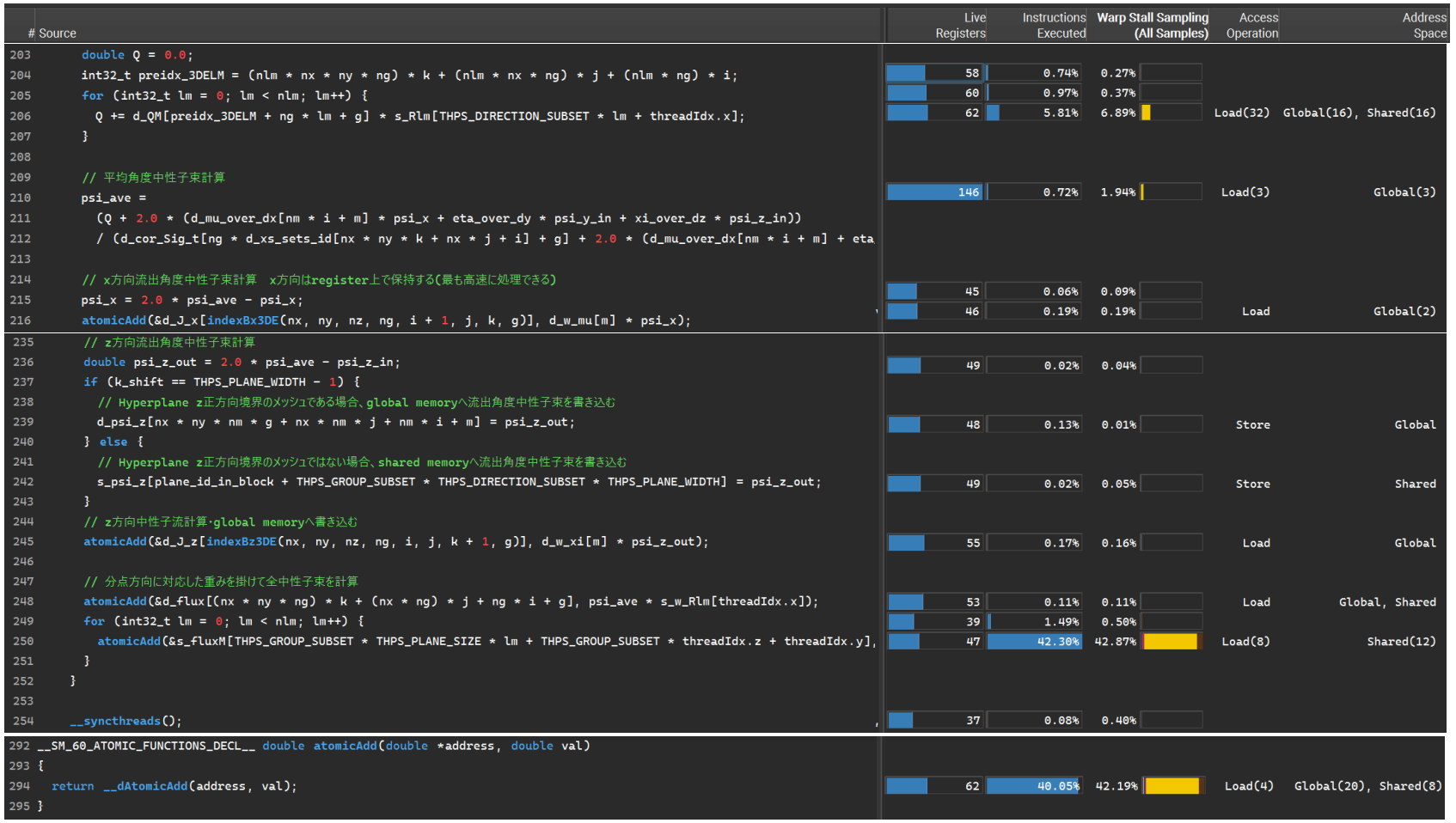
NMh=16	NGh					
	1	2	4	8	16
NDh 1						
2						
4				858.9		
8			1353			
16		2399				
32	3121	3775				

図 3.27 tiled hyperplane transport sweep　version 2 計算時間 [ms]

$NX = NY = NZ = 32, NG = 172, ND = 1200, NL = 7$、第 0 象限のみ

NVIDIA Nsight Compute によるプロファイリング結果 (繰り返し回数: 5)

青←計算時間小　計算時間大→赤



#	Source	Live Registers	Instructions Executed	Warp Stall Sampling (All Samples)	Access Operation	Address Space
203	`double Q = 0.0;`					
204	`int32_t preidx_3DELM = (nlm * nx * ny * ng) * k + (nlm * nx * ng) * j + (nlm * ng) * i;`	58	0.74%	0.27%		
205	`for (int32_t lm = 0; lm < nlm; lm++) {`	60	0.97%	0.37%		
206	`  Q += d_QM[preidx_3DELM + ng * lm + g] * s_Rlm[THPS_DIRECTION_SUBSET * lm + threadIdx.x];`	62	5.81%	6.89%	Load(32)	Global(16), Shared(16)
207	`}`					
208						
209	`// 平均角度中性子束計算`					
210	`psi_ave =`	146	0.72%	1.94%	Load(3)	Global(3)
211	`  (Q + 2.0 * (d_mu_over_dx[nm * i + m] * psi_x + eta_over_dy * psi_y_in + xi_over_dz * psi_z_in))`					
212	`  / (d_cor_Sig_t[ng * d_xs_sets_id[nx * ny * k + nx * j + i] + g] + 2.0 * (d_mu_over_dx[nm * i + m] + eta`					
213						
214	`// x方向流出角度中性子束計算　x方向はregister上で保持する(最も高速に処理できる)`					
215	`psi_x = 2.0 * psi_ave - psi_x;`	45	0.06%	0.09%		
216	`atomicAdd(&d_J_x[indexBx3DE(nx, ny, nz, ng, i + 1, j, k, g)], d_w_mu[m] * psi_x);`	46	0.19%	0.19%	Load	Global(2)
235	`// z方向流出角度中性子束計算`					
236	`double psi_z_out = 2.0 * psi_ave - psi_z_in;`	49	0.02%	0.04%		
237	`if (k_shift == THPS_PLANE_WIDTH - 1) {`					
238	`  // Hyperplane z正方向境界のメッシュである場合、global memoryへ流出角度中性子束を書き込む`					
239	`  d_psi_z[nx * ny * nm * g + nx * nm * j + nm * i + m] = psi_z_out;`	48	0.13%	0.01%	Store	Global
240	`} else {`					
241	`  // Hyperplane z正方向境界のメッシュではない場合、shared memoryへ流出角度中性子束を書き込む`					
242	`  s_psi_z[plane_id_in_block + THPS_GROUP_SUBSET * THPS_DIRECTION_SUBSET * THPS_PLANE_WIDTH] = psi_z_out;`	49	0.02%	0.05%	Store	Shared
243	`}`					
244	`// z方向中性子流計算･global memoryへ書き込む`					
245	`atomicAdd(&d_J_z[indexBz3DE(nx, ny, nz, ng, i, j, k + 1, g)], d_w_xi[m] * psi_z_out);`	55	0.17%	0.16%	Load	Global
246						
247	`// 分点方向に対応した重みを掛けて全中性子束を計算`					
248	`atomicAdd(&d_flux[(nx * ny * ng) * k + (nx * ng) * j + ng * i + g], psi_ave * s_w_Rlm[threadIdx.x]);`	53	0.11%	0.11%	Load	Global, Shared
249	`for (int32_t lm = 0; lm < nlm; lm++) {`	39	1.49%	0.50%		
250	`  atomicAdd(&s_fluxM[THPS_GROUP_SUBSET * THPS_PLANE_SIZE * lm + THPS_GROUP_SUBSET * threadIdx.z + threadIdx.y],`	47	42.30%	42.87%	Load(8)	Shared(12)
251	`}`					
252	`}`					
253						
254	`__syncthreads();`	37	0.08%	0.40%		
292	`__SM_60_ATOMIC_FUNCTIONS_DECL__ double atomicAdd(double *address, double val)`					
293	`{`					
294	`return __dAtomicAdd(address, val);`	62	40.05%	42.19%	Load(4)	Global(20), Shared(8)
295	`}`					

図 3.28 tiled hyperplane transport sweep　version 2　コード解析結果(一部抜粋)

$NX = NY = NZ = 32, NG = 172, ND = 1200, NL = 7$、第 0 象限のみ、$NM_h = 16, NG_h = 8, ND_h = 4$、D-major

NVIDIA Nsight Compute によるプロファイリング結果(繰り返し回数: 5)

図 3.27 より、version 2 コードは version 1 に比べて性能を改善できておらず、どのパラメータの組み合わせに対しても悪化していることが分かる。この原因を図 3.28 のコード解析結果から考察する。

図 3.28 より、250 行目の演算と、294 行目の `atomicAdd()`などに関連する warp stall がさらに増加していることが分かる。warp stall の要因は大半が Short Scoreboard に占められており、shared memory への過負荷と bank conflict を生じ得る非効率的なアクセスが原因と考えられる。ただし 250 行目と 294 行目については、解析の都合上 atomicAdd()による stall とそれ以外の shared memory 操作に伴う stall が分離できていなものの、別途実施した CUDA における中間コード SASS の解析結果では、`atomicAdd()`による影響がより大きいが示されている。

250 行目については、図 3.29 に示すようにひどい場合で 20-way bank conflict を生じており、37.4 B 回(B=Billion)のアクセスが発生している。プロファイリングによると、そのうち 6.69 B 回のアクセスは余剰なアクセスであり、理想的なアクセス回数は 30.7 B 回である。なお、余剰アクセス回数以前に、そもそもアクセス回数が多いのは `atomicAdd()`が呼び出されていることに起因する。

#	Source	L1 Conflicts Shared N-Way	L1 Wavefronts Shared Excessive	L1 Wavefronts Shared	L1 Wavefronts Shared Ideal
250	atomicAdd(&s_fluxM[THPS_GROUP_SUBSET *	20	6.69B	37.4B	30.7B

図 3.29　version 2 コード 250 行目 shared memory アクセスの解析結果

以上を踏まえると、shared memory に依存する計算を増やしすぎたためアクセス回数が多すぎることと、さらにその操作が `atomicAdd()`であることや bank conflict を生じやすいものになってしまったことが性能悪化の原因と考えられる。

そこで、version 3 では shared memory に対する `atomicAdd()`呼び出し回数の削減を試みる。

c) version 3

version 3 コードでは、shared memory で$\phi^{l}_{m,g,i,j,k}$を合算するときに、同じgを担当する thread が 1 ずつlmをずらすことで、shared memory に対する atomic 操作を不要としている。ただし、実装の都合上、本手法が利用できるのは$(NL+1)^2 \geq ND_{\mathrm{sub}}$の場合のみであり、warp 内 thread の動作が同期していることを前提としている。さらに、block 内の warp 同士の計算、操作に依存性がない、つまり同じをg, idxを担当する thread が 1 warp に収まっていることも前提となる。そのため、パラメータ設定に注意を要する。具体的には、D-major の場合はND_hが、G-major の場合はNG_hが 32 の約数でなければならない。

この場合、各スレッドは以下のように動作する。

1. 各スレッドについて、$lm = d_{\mathrm{sub}}$ (d_{sub}は block 内での飛行方向の番号、スレッドごとに異なる)
2. $\psi_{g,d,i,j,k} \times w_d \times R_l^m$を各展開次数$(l,m)$について計算する
3. 2.の結果を shared memory の$\phi^{l}_{m,g,i,j,k}$に加算する (**atomic ではない**)
4. Warp を同期 (`__syncwarp()`を呼び出す)
   ※Computation Capability 7.x 以降、暗黙的な warp 同期は安全ではない
5. $lm = lm + 1$
6. $d_{\mathrm{sub}} = 0$かつ$lm = (NL+1)^2$、または$d_{\mathrm{sub}} \neq 0$かつ$lm = d_{\mathrm{sub}} - 1$なら 7.へ。
7. $lm = (NL+1)^2$のとき、$lm = 0$
8. 2.へ戻る
9. block 内のスレッドを同期 (block 内の shared memory に対する操作の完了を待つ)
10. $lm = 0$
11. (同じgを担当するスレッドのうち 1 スレッドのみ動作 他のスレッドは休眠)
    shared memory の$\phi^{l}_{m,g,i,j,k}$を global memory の$\phi^{l}_{m,g,i,j,k}$へ加算する(global memory に対する `atomicAdd`)
12. $lm = lm + 1$、$lm = (NL+1)^2$のとき 9.へ
13. 次の hyperplane へ

この手法により、shared memory に対する `atomicAdd()`は不要で、かつ global memory に対する `atomicAdd()`の回数を version 1 に比べ$1/ND_h$に減らすことができる。展開次数が大きくなるほど、この効果は顕著になると考えられる。

$G_h = 2, ND_h = 8, NM_h = 16, NL = 2$, D-major の場合における各 thread の動作イメージを図 3.30 に示す。図から、warp 内 thread が同期していれば、shared memory へのアクセス競合が起こらず、不可分操作が不要であることが分かる。図 3.30 に示された block は `blockIdx.x=blockIdx.y=blockIdx.z=0` の block である。また、このとき図 3.30 に示すようにlmについて連続となるようにϕ^{l}_{m}を shared memory に格納する必要がある。

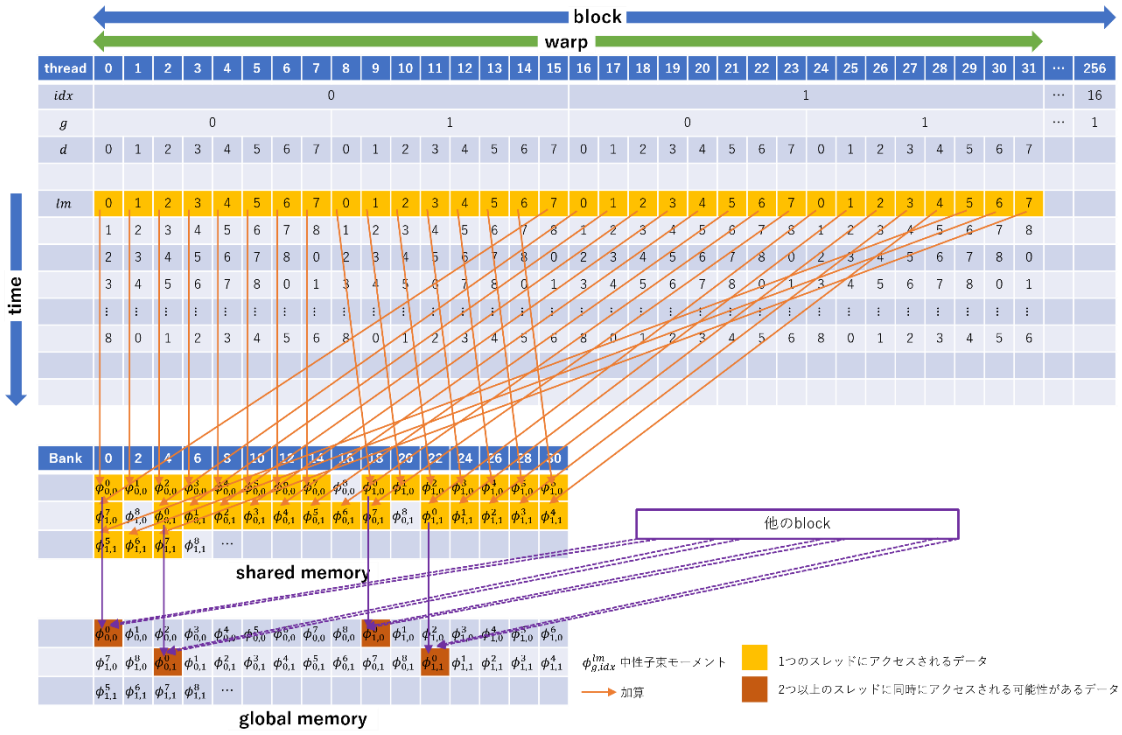


図 3.30 version 3 ループ 0 回目の動作イメージ ($NL = 2$)

ここで、version 1, 2 と同様に、パラメータNM_h, NG_h, ND_hの組み合わせのいくつかで version 3 コードによりテスト体系について transport sweep を実施した。ただし解析対象は第 0 象限飛行方向について計算する kernel のみとする。計算時間の測定結果を図 3.31 に示す。また、そのうち 1 つの場合におけるコードの解析結果を図 3.32 に示す。また、図 3.25、図 3.28、図 3.32 で示した解析について、特に warp の状態を比較したものを図 3.33 に示す。

GmajV3

NMh=64		NGh				
		1	2	4	8	16
NDh	1					
	2					
	4					
	8		759.79			
	16	750.82				
	32					

NMh=16		NGh				
		1	2	4	8	16
NDh	1					
	2					
	4					
	8				350.16	
	16			347.18		
	32	580.87	359.3			

DmajV3

NMh=64		NGh				
		1	2	4	8	16
NDh	1					
	2					
	4					
	8		323.15			
	16	396.15				
	32					

NMh=16		NGh				
		1	2	4	8	16
NDh	1					
	2		2739			
	4		1035			
	8	1400	291.32	319.18	390.23	
	16	438.66	305.39	377.26		
	32	307.77	423.37			

図 3.31 tiled hyperplane transport sweep　version 3 計算時間 [ms]

$NX = NY = NZ = 32, NG = 172, ND = 1200, NL = 7$、第 0 象限のみ

NVIDIA Nsight Compute によるプロファイリング結果 (繰り返し回数: 5)

青←計算時間小　計算時間大→赤

#	Source	Live Registers	Instructions Executed	Warp Stall Sampling (All Samples)	Access Operation	Address Space
203	`double Q = 0.0;`					
204	`int32_t preidx_3DELM = (nlm * nx * ny * ng) * k + (nlm * nx * ng) * j + (nlm * ng) * i;`	60	1.60%	0.78%		
205	`for (int32_t lm = 0; lm < nlm; lm++) {`	59	2.10%	1.86%		
206	`Q += d_QM[preidx_3DELM + ng * lm + g] * s_Rlm[THPS_DIRECTION_SUBSET * lm + threadIdx.x];`	62	12.61%	20.85%	Load(32)	Global(16), Shared(16)
207	`}`					
208						
209	`// 平均角度中性子束計算`					
210	`psi_ave =`	150	1.57%	5.06%	Load(3)	Global(3)
211	`(Q + 2.0 * (d_mu_over_dx[nm * i + m] * psi_x + eta_over_dy * psi_y_in + xi_over_dz * psi_z_in))`					
212	`/ (d_cor_Sig_t[ng * d_xs_sets_id[nx * ny * k + nx * j + i] + g] + 2.0 * (d_mu_over_dx[nm * i + m] +`					
247	`// 分点方向に対応した重みを掛けて全中性子束を計算`					
248	`atomicAdd(&d_flux[(nx * ny * ng) * k + (nx * ng) * j + ng * i + g], psi_ave * s_w_Rlm[threadIdx.x]);`	53	0.32%	0.23%	Load	Global, Shared
249	`}`					
250						
251	`{`					
252	`// Warp内のスレッドごとにlmをずらすことでatomic演算を削減`					
253	`int32_t lm = threadIdx.x;`					
254	`while (true) {`					
255	`if (is_active) s_fluxM[nlm * THPS_GROUP_SUBSET * threadIdx.z + nlm * threadIdx.y + lm] += psi_ave * s_`	43	17.56%	25.88%	Load(2)…	Shared(3)
256	`__syncwarp();`	147	2.93%	0.39%		
257	`if (lm == (threadIdx.x == 0 ? nlm - 1 : threadIdx.x - 1)) break;`	43	2.97%	0.67%		
258	`lm++;`	43	2.93%	0.80%		
259	`if (lm == nlm) lm = 0;`	43	26.34%	26.49%	Load(2)…	Shared(3)
260	`}`					
261	`}`					
262						
263	`__syncthreads();`	39	0.23%	1.07%		
264						
265	`if (is_active) {`	39	0.14%	0.02%		
266	`// block内で加算したデータをglobalへ`					
267	`if (threadIdx.x == 0) {`					
268	`for (int32_t lm = 0; lm < nlm; lm++) {`	60	1.83%	0.17%		
269	`atomicAdd(&d_fluxM[(nlm * nx * ny * ng) * k + (nlm * nx * ng) * j + (nlm * ng) * i + ng * lm + g],`	62	7.13%	3.00%	Load(16)	Global(3), Shared(16)
270	`s_fluxM[nlm * THPS_GROUP_SUBSET * threadIdx.z + nlm * threadIdx.y + lm] = 0.0;`	61	3.11%	2.06%	Store(1…	Shared(16)
271	`}`					
272	`}`					
273	`}`					
292	`__SM_60_ATOMIC_FUNCTIONS_DECL__ double atomicAdd(double *address, double val)`					
293	`{`					
294	`return __dAtomicAdd(address, val);`	62	6.08%	3.58%		Global(20)
295	`}`					

図 3.32 tiled hyperplane transport sweep　version 3　コード解析結果(一部抜粋)

$NX = NY = NZ = 32, NG = 172, ND = 1200, NL = 7$、第 0 象限のみ、$NM_h = 16, NG_h = 2, ND_h = 8$、D-major

NVIDIA Nsight Compute によるプロファイリング結果(繰り返し回数: 5)

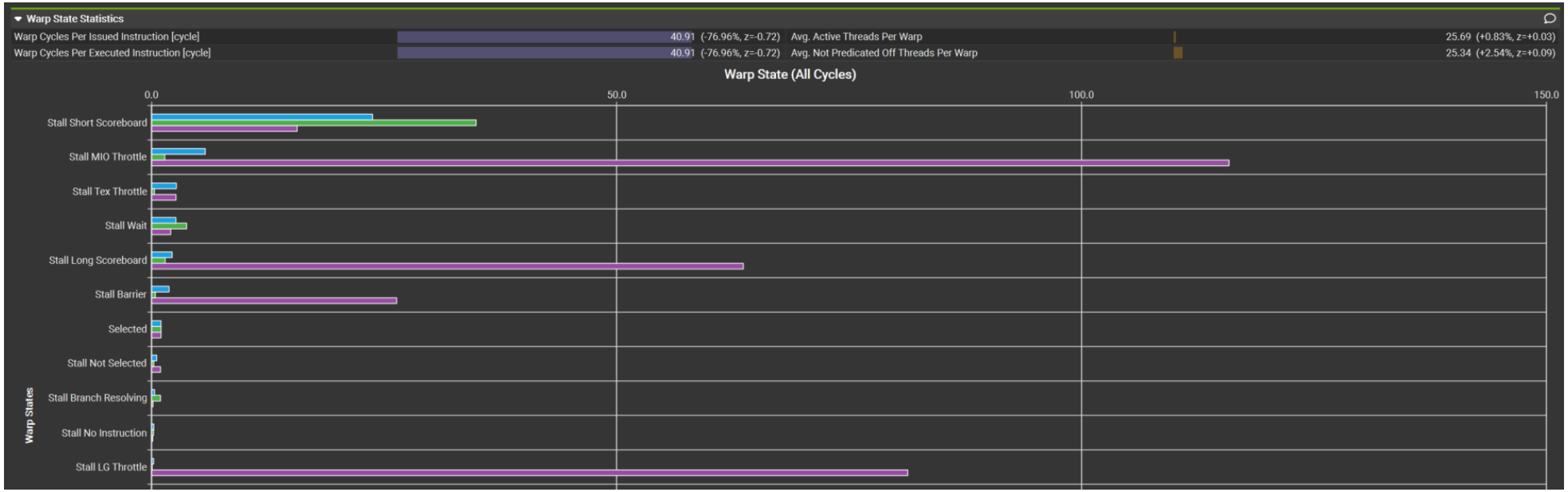


図 3.33 version 1, 2, 3 コードにおける warp 状態の解析結果の比較

NVIDIA Nsight Compute によるプロファイリング結果(繰り返し回数: 5)

紫：version 1、緑：version 2、青：version 3

図 3.31 より、version 1, 2 に比べ version 3 では多くのケースで計算時間を短縮できていることがわかる。各 version の最短計算時間を以下の表に示す。

表 3.2 各 version における最短計算時間とパラメータ

	パラメータ		計算時間 [ms]	高速化率
	G-major / D-major	(NM_h, NG_h, ND_h)		
version 1	D-major	$(64, 8, 2)$	822.25	1.00
version 2	D-major	$(16, 8, 4)$	858.90	0.96
version 3	D-major	$(16, 2, 8)$	291.32	2.82

version 3 で高速化に成功した要因は、global memory に対する `atomicAdd()`の呼び出しを version 2 と同程度に減らしたまま、shared memory に対する `atomicAdd()`の呼び出しを無くすことができたことが大きく寄与していると考えられる。図 3.32 の 294 行目からわかるように、`atomicAdd()`による warp stall は全体の数%にとどまっており、version 1, 2 に比べ寄与が大きく減少している。図 3.33 からも、version 3 では warp stall を大きく減らすことができていることが分かる。

一方で、206, (255,)259 行目の Short Scoreboard が全体の warp stall のうち多くを占めている。ただし解析の都合上、259 行目の解析結果は 255 行目のものを内包している。例えば、259 行目では 17.1 B 回のメモリアクセスが発生しており、そのうち 6.85 B は非効率なアクセスによる余剰アクセスであると解析されており、Short Scoreboard を改善するには、shared memory へのアクセスの多さと非効率さを改善する必要があると考えられる。

以上のように、version 3 コードでもメモリアクセスに改善の余地があることが解析結果よりうかがえるが、この時点で GPU の倍精度(FP64)計算性能のうち約 80.9 %を使用していることが解析結果より判明している。加えて、第 4 章で述べるように version 3 時点で transport sweep 以外の拡散加速計算などの計算が計算時間の大半を占めるようになっており、transport sweep コードのこれ以上の改善は優先度が低いと判断した。よって、本研究では version 3 コードをα固有値計算に利用することとし、これ以上の改善は行わない。

3.4.4　消費メモリ量

tiled hyperplane transport sweep version 3 コードを用いたS_N法に基づくα固有値計算において要求されるメモリ量の概算を以下に示す。ただし、断面積データ等の定数に必要な領域やループ変数などの自動変数により消費されるメモリは考慮しない。

a)　global memory

global memory には以下のような領域が求められる。

- ◆　ψ_d, Q_dは global memory に保持しない。
- ◆　$\psi_d^{\mathrm{xin/out}}$は register で保持し、global memory は使用しない。
- ◆　$\psi_d^{\mathrm{yin/out}}, \psi_d^{\mathrm{zin/out}}$は hyperplane の$y, z$方向境界面の必要分だけ確保する。

以上を参考に、計算に必要なデータを保持するために要求されるメモリ量が以下のように概算できる。

$$\mathrm{mem}(\phi_l^m) = \mathrm{sizeof(REAL)} \times (NX \times NY \times NZ) \times NG \times (NL+1)^2 \tag{3.1}$$

$$\mathrm{mem}(Q_l^m) = \mathrm{sizeof(REAL)} \times (NX \times NY \times NZ) \times NG \times (NL+1)^2 \tag{3.2}$$

$$\mathrm{mem}(\psi^{y\mathrm{out}}) = \mathrm{sizeof(REAL)} \times (NX \times HW) \times NG \times ND \tag{3.3}$$

$$\mathrm{mem}(\psi^{z\mathrm{out}}) = \mathrm{sizeof(REAL)} \times (NX \times NY) \times NG \times ND \tag{3.4}$$

$$\mathrm{mem}(J^{x\mathrm{out}}) = \mathrm{sizeof(REAL)} \times (NX+1) \times NY \times NZ \times NG \tag{3.5}$$

$$\mathrm{mem}(J^{y\mathrm{out}}) = \mathrm{sizeof(REAL)} \times NX \times (NY+1) \times NZ \times NG \tag{3.6}$$

$$\mathrm{mem}(J^{z\mathrm{out}}) = \mathrm{sizeof(REAL)} \times NX \times NY \times (NZ+1) \times NG \tag{3.7}$$

mem(·)　：データ保持に要求されるメモリ量
sizeof(·)　：任意の変数型のデータサイズ
REAL　：任意の実数型　sizeof(float) = 4 byte, sizeof(double) = 8 byte
NX　：x方向のメッシュ分割数
NY　：y方向のメッシュ分割数
NZ　：z方向のメッシュ分割数
HW　: hyperplane の 1 辺のメッシュ数 (hyperplane width)　$NM_h = HW^2$
NG　：エネルギー群数
NL　：非等方散乱次数上限
ND　：飛行方向分割数

ここで、例として$NXNYNZ = 32^3, HW = 4, NG = 172, NL = 3, ND = 72$の場合の要求量例を表 3.3 に示す。ただし、1 MB $= 2^{20}$ byteとする。

表 3.3 要求 global memory 量の例
(倍精度、$NXNYNZ = 32^3, HW = 4, NG = 172, NL = 3, ND = 72$)

変数	要求メモリ量 [MB]
ϕ_l^m	688.0
Q_l^m	688.0
ψ^{yout}	12.1
ψ^{zout}	96.8
J^{xout}	44.3
J^{yout}	44.3
J^{zout}	44.3
計	1630.0

global memory 消費量が約 1.6 GB に対して、計算に使用する GPU の搭載メモリ容量 24 GB は十分に大きいことから、$NXNYNZ = 32^3, HW = 4, NG = 172, NL = 3, ND = 72$の条件下における計算は現実的に可能であるといえる。

b) shared memory

shared memory には以下のような領域が求められる。

- ϕ_l^mを block 内で保持するためのメモリ
- $\psi_d^{\mathrm{yin/out}}, \psi_d^{\mathrm{zin/out}}$の block 内やり取りのためのメモリ
- $R_l^m, w_d R_l^m$をあらかじめ計算したデータを格納するためのメモリ

以上を参考に、計算に必要なデータを保持するために要求されるメモリ量が以下のように概算できる。

$$\mathrm{mem}(\phi_l^m) = \mathrm{sizeof(REAL)} \times NM_h \times NG_h \times (NL+1)^2 \tag{3.8}$$

$$\mathrm{mem}\left(\psi_d^{\mathrm{yin/out}}\right) = \mathrm{sizeof(REAL)} \times NM_h \times NG_h \times ND_h \tag{3.9}$$

$$\mathrm{mem}\left(\psi_d^{\mathrm{zin/out}}\right) = \mathrm{sizeof(REAL)} \times NM_h \times NG_h \times ND_h \tag{3.10}$$

$$\mathrm{mem}(R_l^m) = \mathrm{sizeof(REAL)} \times ND_h \times (NL+1)^2 \tag{3.11}$$

$$\mathrm{mem}(w_d R_l^m) = \mathrm{sizeof(REAL)} \times ND_h \times (NL+1)^2 \tag{3.12}$$

$NM_h = 16, HM = 8, NG = 172, NL = 3, ND = 72$の場合の要求量例を表 3.4 に示す。ただし、1 KB $= 2^{10}$ byteとする。

表 3.4 block あたりの要求 shared memory 量の例
(倍精度、$NM_h = 16, NG_h = 2, ND_h = 8, NG = 172, NL = 3, NM = 72$)

変数	要求メモリ量 [KB]
ϕ_l^m	32.0
$\psi_d^{\mathrm{yin/out}}$	2.0
$\psi_d^{\mathrm{zin/out}}$	2.0
R_l^m	1.0
$w_d R_l^m$	1.0
計	38.0

なお、shared memory / L1 cache (RTX4090: 128 KB/SM)は同一のオンチップメモリを両者で共用しており、shared memory を多く使用すると L1 Cache への割当がその分減少することと、SM・block あたりの shared memory には上限(100 KB 前後)があることに注意する必要がある。

3.5　GPGPU を用いたα固有値拡散加速計算の実装

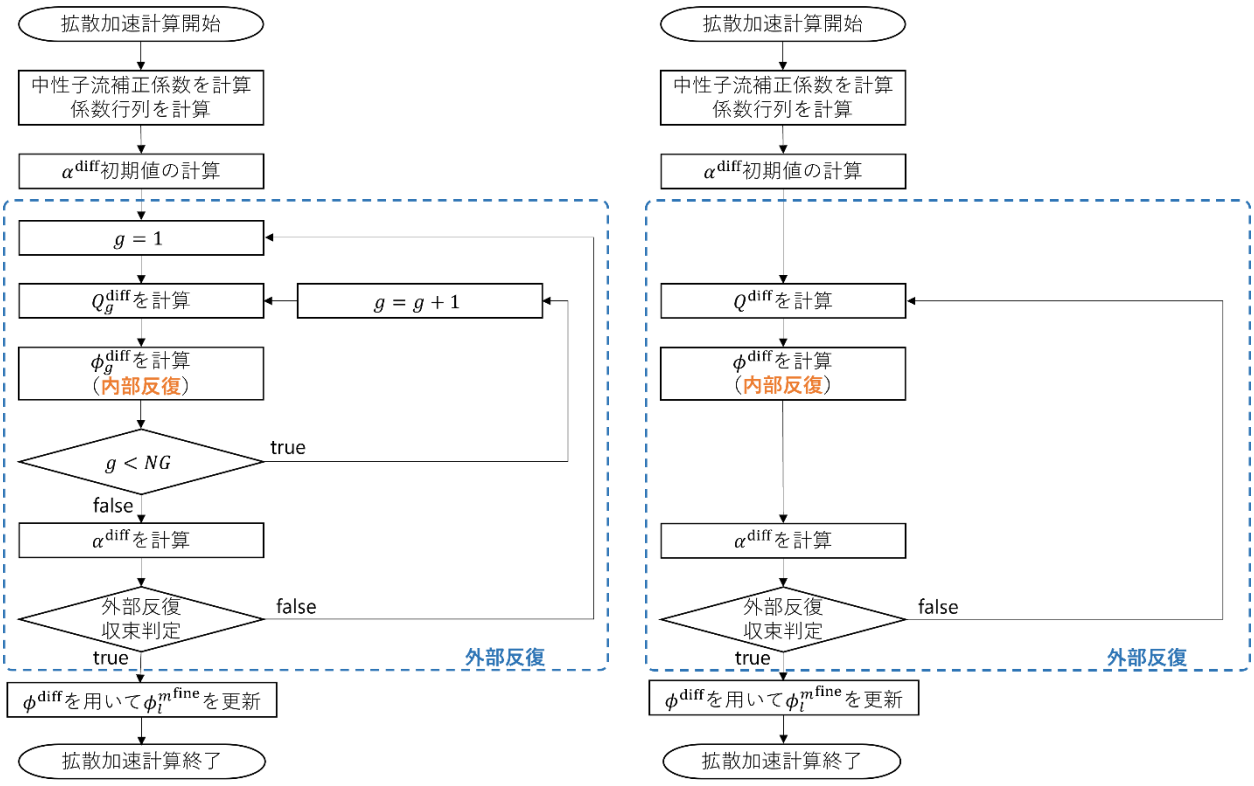
本節では GPGPU を用いたα固有値拡散加速計算の実装について述べる。

3.5.1 項では、α固有値拡散加速計算の計算フローを示し、3.5.2 項では GPGPU を用いた具体的な実装について述べる。3.5.3 項では、計算に必要な global memory の量を概算する。

3.5.1　計算フロー

第 2 章の理論に基づいた、α固有値拡散加速計算の計算フローを図 3.34 に示す。S_N法と同様に、外部反復にガウスザイデル法を用いる場合とヤコビ法を用いる場合があるが、並列化度を大きくするため、本研究ではS_N法と同様にヤコビ法を採用する。

本研究で並列化の対象とするのは、$Q^{\mathrm{diff}}, \phi^{\mathrm{diff}}, \alpha^{\mathrm{diff}}$の計算とする。



(i) 外部反復にガウスザイデル法を用いる場合　　(ii) 外部反復にヤコビ法を用いる場合

図 3.34 α固有値拡散加速計算の計算フロー

3.5.2　計算の並列化

拡散加速計算に必要なほとんどの計算は行列・ベクトル積など、線形代数の形で表現でき、多くの計算に線形代数ライブラリを利用することができる。GPU に対応した線形代数ライブラリはいくつか存在するが、本研究では cuBLAS[13]及び MAGMA[14],[15]を利用する。

以降では、並列化の対象とする$\alpha^{\mathrm{diff}}, Q^{\mathrm{diff}}, \phi^{\mathrm{diff}}$の計算の実装についてそれぞれ述べる。

a)　α^{diff}の計算

α^{diff}は式(2.75)により更新でき、これはベクトル$\phi_{g,i,j,k}^{\mathrm{diff},(n')}/v_g$とベクトル$\Delta x_i \Delta y_j \Delta z_k$の内積の逆数を計算することに等しい。ただし、ベクトル$\Delta x_i \Delta y_j \Delta z_k$はエネルギー$NG$群の分だけ複製し結合された、$NX \times NY \times NZ \times NG$個の要素を持つベクトルとする。

$$\alpha^{\mathrm{diff},(n')} = \frac{1}{\sum_{g=1}^{NG}\sum_{k=1}^{NZ}\sum_{j=1}^{NY}\sum_{i=1}^{NX}\frac{\phi_{g,i,j,k}^{\mathrm{diff},(n')}}{v_g}\Delta x_i \Delta y_j \Delta z_k} \quad (2.75)$$

再掲

$\phi_{g,i,j,k}^{\mathrm{diff}}/v_g$の計算は$NX \times NY \times NZ \times NG$ thread に並列化した簡単な kernel として実装できる。ただし、積演算に比べて除演算は高いコストを要するため、$\phi_{g,i,j,k}^{\mathrm{diff}}$を$v_g$で除算するのではなく、あらかじめ計算した逆数$1/v_g$を乗算するように実装する。ごく単純な実装であるため、実装の詳細は省略する。また、ベクトルの内積には GPU 対応の線形代数ライブラリの一つである cuBLAS が提供する `cublasDdot_v2()`関数などを利用する。

b)　Q^{diff}の計算

中性子源Q^{diff}の計算は、S_N法における非等方散乱中性子源Q_l^mの計算において展開次数上限を$NL = 0$としたときの計算と等価となる。よって、3.4.2 項で説明した実装をほぼそのまま用いて中性子源Q^{diff}の計算の計算を実装できるため、実装の詳細は省略する。

c) ϕ^{diff}の計算

ϕ^{diff}の計算では、連立一次方程式(2.66)を解く必要がある。

$$A^{x-}_{g,i,j,k}\phi_{g,i-1,j,k} + A^{y-}_{g,i,j,k}\phi_{g,i,j-1,k} + A^{z-}_{g,i,j,k}\phi_{g,i,j,k-1} + A^{0}_{g,i,j,k}\phi_{g,i,j,k} + A^{x+}_{g,i,j,k}\phi_{g,i+1,j,k} + A^{y+}_{g,i,j,k}\phi_{g,i,j+1,k} + A^{z+}_{g,i,j,k}\phi_{g,i,j,k+1} = Q_{g,i,j,k} \quad (2.66)$$

再掲

連立一次方程式の数値解法には、LU 分解法のような直接解法や、ヤコビ法やガウスザイデル法、LU 分解法を発展させた ADI 法などの定常反復法、そして GMRES 法(一般最小残差法)や BiCGSTAB 法(安定化双共役勾配法)[16]などの非定常反復法が知られている。しかし、LU 分解法は逐次的な計算を伴うため並列化できないことに加え、巨大で疎な係数行列**A**を分解した行列**L, U**を保持するために膨大なメモリを要することから、一般的に拡散加速計算に用いられない。ガウスザイデル法、ADI 法も逐次的処理を含むため、並列化との親和性が低いとされる。一方で、ヤコビ法や GMRES 法、BiCGSTAB 法などの反復法は直接解法に比べメモリを節約でき、並列化との親和性も高いことから、拡散加速計算に用いられることが多い。本研究では、非定常反復法の中でも計算速度や安定性に優れるとされる BiCGSTAB 法を採用する。

BiCGSTAB 法などの反復法においては、収束性改善のための前処理が一般に必要とされ、代表的な前処理手法としては点ヤコビ前処理や不完全 LU 分解法が知られている。点ヤコビ前処理は、係数行列**A**の対角成分のみを取り出したものを前処理行列**M**とするもので、単純でありながら、係数行列**A**が対角行列に近い場合は優れた性能が得られるとされる。不完全 LU 分解法はメモリ節約のために係数行列**A**の 0 成分を 0 に保つようにした、不完全な LU 分解を利用する手法である。しかし、逐次処理を伴うことは LU 分解法と変わらないため、並列化との親和性は低い。よって、本研究では点ヤコビ前処理を利用する。

GPGPU に対応した BiCGSTAB 法の実装は MAGMA ライブラリにより提供されている。点ヤコビ前処理にも対応しているため、計算コードの実装にはこれを利用する。

3.5.3　消費メモリ量

以上で述べたように実装したα固有値拡散加速計算において要求される global memory 量の概算を以下に示す。ただし、断面積データ等の定数に必要な領域やループ変数などの自動変数により消費されるメモリは考慮しない。また、shared memory についてはライブラリによる要求量が公開されていないためここでは示さない。前処理付き BiCGSTAB が要求するメモリ量は実装により異なるが、表 3.5 では文献[16]で使用されている 9 個のベクトルを保持するのに必要な量を示す。ただし、$1\ \mathrm{MB} = 2^{20}$ byteとする。

$$\mathrm{mem}(\phi) = \mathrm{sizeof(REAL)} \times (NX \times NY \times NZ) \times NG \tag{3.13}$$

$$\mathrm{mem}(Q) = \mathrm{sizeof(REAL)} \times (NX \times NY \times NZ) \times NG \tag{3.14}$$

$$\mathrm{mem}(D_{\mathrm{cor}}^{x}) = \mathrm{sizeof(REAL)} \times (NX + 1) \times NY \times NZ \times NG \tag{3.15}$$

$$\mathrm{mem}\left(D_{\mathrm{cor}}^{y}\right) = \mathrm{sizeof(REAL)} \times NX \times (NY + 1) \times NZ \times NG \tag{3.16}$$

$$\mathrm{mem}(D_{\mathrm{cor}}^{z}) = \mathrm{sizeof(REAL)} \times NX \times NY \times (NZ + 1) \times NG \tag{3.17}$$

$$\begin{aligned} &\mathrm{mem}(\mathbf{A}) \\ &= \mathrm{sizeof(REAL)} \times NNZ + \mathrm{sizeof(INT)} \times (NNZ + NX \times NY \times NZ \times NG) \\ &= \mathrm{sizeof(REAL)} \times \begin{pmatrix} 7 \times NX \times NY \times NZ \\ -2 \times NX \times NY - 2 \times NY \times NZ - 2 \times NZ \times NX \end{pmatrix} NG \\ &\qquad + \mathrm{sizeof(INT)} \times (NNZ + NX \times NY \times NZ \times NG) \end{aligned} \tag{3.18}$$

$$\mathrm{mem(PBiCGSTAB)} = \mathrm{sizeof(REAL)} \times (9 \times NX \times NY \times NZ \times NG) \tag{3.19}$$

NNZ　　：係数行列の非零要素数
INT　　：任意の整数型 sizeof(int32_t) = 4 byte
PBiCGSTAB　：前処理付き BiCGSTAB

表 3.5 要求 global memory 量の例
(倍精度、$NXNYNZ = 32^3, NG = 172$)

変数	要求メモリ量 [MB]
ϕ	43.0
Q	43.0
D_{cor}^{x}	44.3
D_{cor}^{y}	44.3
D_{cor}^{z}	44.3
A	460.9
PBiCGSTAB	387.0
計	1066.9

S_Nに基づくα固有値計算に必要な量と合わせた global memory 消費量が約 2.6 GB であるのに対して、計算に使用する GPU の搭載メモリ容量 24 GB は十分に大きいことから、この条件下における計算は現実的に可能であるといえる。

3.6 本章のまとめ

本章では、GPGPU を用いたS_N法に基づくα固有値計算コードの開発について述べた。

3.2 節では、GPU を科学技術計算などに応用する技術である GPGPU の概要を説明した。3.2.1 項では、host(CPU)と device(GPU)がそれぞれ別のメモリを持ち、device は kernel と呼ばれる全 thread に共通したコードを実行するという CUDA プログラミングモデルについて説明した。3.2.2 項では、CUDA における grid, block, warp からなる thread の階層構造と、32 thread を 1 warp とし、warp 内 thread は同一命令を同時に実行する SIMT アーキテクチャについて説明した。3.2.3 項では CUDA における主に global, shared memory からなるメモリの階層構造と、メモリアクセスの仕組みとメモリアクセス時に意識すべき点について述べた。

3.3 節では、GPGPU を用いた核計算コード開発において一般に課題となる事柄について述べた。例えば、エネルギー群インデックスgについてデータと thread がそれぞれ連続になるよう構成することで容易に coalesced access を実現できることを説明した。一方で、GPGPU を利用したコードでは coalesced access の他にもパフォーマンスに影響する要素が多くあり、NVIDIA Nsight Compute などのプロファイラーなどを用いて問題点を特定、改善しながら開発をする必要があることを述べた。

3.4 節では、GPGPU を用いた、S_N法に基づくα固有値計算の実装について述べた。3.4.1 項では全体の計算フローを示し、3.4.2 項では効率的なメモリアクセスを意識した非等方散乱中性子源計算の実装手法を示した。3.4.3 項では、transport sweep の実装手法を示した。GPU の性能を十分に利用するために、transport sweep を空間メッシュについて並列化する手法である tiled hyperplane transport sweep を用いることを述べた。計算中に必要となるが、高コス

トを要する global memory に対する `atomicAdd()`関数の呼び出しを減らすため、version 1, 2, 3 とコードを改善する手法を示した。version 3 では、shared memory を利用したうえで、各 thread が計算する展開次数(l, m)の組をずらして操作の競合を回避することで必要な `atomicAdd()`関数の呼び出しを削減した。そして、性能測定の結果、本修論研究では優れた性能を示した version 3 を採用したことを述べた。3.4.4 項では計算に必要な global memory と shared memory の量を概算した。典型的な計算条件下において global memory 消費量は搭載メモリ容量の 1/10 以下、block あたりの shared memory 消費量は上限の約 1/3 であり、メモリ消費量が現実的な範囲内に収まっていることを確認した。

3.5 節では、α固有値拡散加速計を GPGPU により実装する手法について述べた。多くの計算は GPU に対応した線形代数ライブラリを用いて実装でき、中性子源の計算には、S_N法の非等方散乱中性子源計算の手法を流用できることを述べた。また、連立一次方程式の解法には、並列化との親和性が高い、点ヤコビ前処理を適用した BiCGSTAB 法を採用することを述べた。3.5.3 項では拡散加速計算に必要な global memory の量を概算し、典型的な計算条件下における消費量はS_N法の計算に必要な量と合わせても、搭載メモリ容量の約 1/9 であり、消費量が実行可能な範囲内に収まっていることを確認した。

3.7 参考文献

[1] “CUDA C++ Programming Guide;” NVIDIA Corporation; https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html; (current as of Dec. 11, 2023).

[2] “ CUDA Toolkit - Free Tools and Training,” NVIDIA Corporation; https://developer.nvidia.com/cuda-toolkit; (current as of Dec. 13, 2023).

[3] “OpenCL - The Open Standard for Parallel Programming of Heterogeneous Systems,” The Khronos Group; https://www.khronos.org//; (current as of Dec. 13, 2023).

[4] “Homepage | OpenACC;” OpenACC Organization; https://www.openacc.org/; (current as of Dec. 13, 2023).

[5] “Home - OpenMP,” OpenMP ARB; https://www.openmp.org/; (current as of Dec. 13, 2023).

[6] “CUDA C++ Best Practices Guide,” NVIDIA Corporation; https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html; (current as of Dec. 20, 2023).

[7] “NVIDIA TESLA V100 GPU Architecture,” NVIDIA Corporation; https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf; (current as of Dec. 13, 2023).

[8] “Whitepaper NVIDIA’s Next Generation CUDA™ Compute Architecture: Fermi™,” NVIDIA Corporation; https://www.nvidia.com/content/pdf/fermi_white_papers/nvidia_fermi_compute_architecture_whitepaper.pdf; (current as of Dec. 20, 2023).

[9] “NVIDIA® Nsight™ Development Platform, Visual Studio Edition 4.7 User Guide: Issue

Efficiency;" NVIDIA Corporation; https://docs.nvidia.com/gameworks/content/developertools/desktop/analysis/report/cudaexperiments/kernellevel/issueefficiency.htm; (current as of Dec. 20, 2023).

[10] "NVIDIA Nsight Compute," NVIDIA Developer; https://developer.nvidia.com/nsight-compute; (current as of Jan. 26, 2024).

[11] S. RENNICH, D. APPELHANS, L GRINBERG, et al., "S_N Transport on Accelerations," DOE CoE Probability Workshop, (2016).

[12] C. GONG, L JIE, C. LIHUA, et al., "GPU Accelerated Simulations of 3D Deterministic Particle Transport Using Discrete Ordinates Method," Journal of Computational Physics 230 15, 6010 (2011); https://doi.org/10.1016/j.jcp.2011.04.010.

[13] "cuBLAS," NVIDIA Corporation; https://developer.nvidia.com/cublas; (current as of Jan. 15, 2024).

[14] "MAGMA;" Innovative Computing Laboratory; https://icl.utk.edu/magma/; (current as of Jan. 15, 2024).

[15] H. ANZT, W. Sawyer, S. TOMOV, et al., "Optimizing Krylov Subspace Solvers on Graphics Processing Units," in 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, pp. 941–949, IEEE, Phoenix, AZ, USA (2014); https://doi.org/10.1109/IPDPSW.2014.107.

[16] H. A. VAN DER VORST, "Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems," SIAM J. Sci. and Stat. Comput. 13 2, 631 (1992); https://doi.org/10.1137/0913035.

第4章　検証計算

4.1　本章の概要

本章では、第 2 章及び第 3 章で述べた内容に基づき開発した、GPGPU を用いたS_N法に基づくα固有値計算コードの妥当性や高速化性能について検証する。

4.2 節では、本章の妥当性確認及び検証における計算条件について述べる。4.3 節では、先行研究によるα実験値と、本研究で開発した CPU 及び GPU コードによるαの計算結果を比較することで、開発したコードの妥当性確認及び検証を実施する。また、熱中性子散乱則に起因するα計算値の不確かさについても評価する。4.4 節では、開発した CPU 及び GPU コードの計算時間を比較し、本研究で開発した GPU コードの有効性を検証する。

4.2　計算条件

本節では、本章の妥当性確認及び検証における計算条件について述べる。なお、本節で示した各種条件については、特筆なき限り、本章で共通のものとする。

まず、開発したコードと比較検証するために利用した先行研究の実験体系について述べる。先行研究[1]における実験装置の概要を図 4.1 に示す。

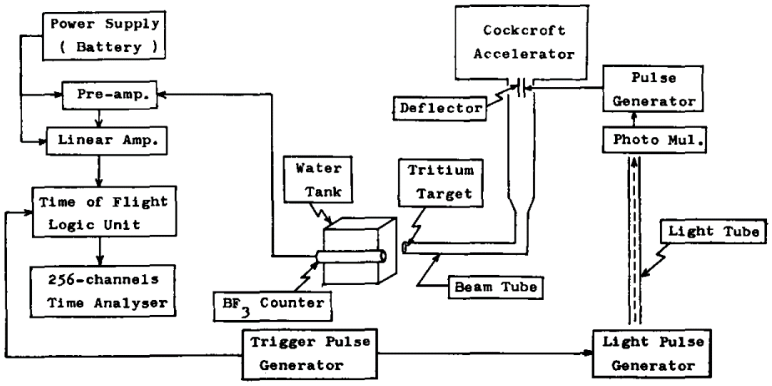


図 4.1 先行研究における実験装置の概要[1]

実験室の室温及び水槽内の水温は 10 ℃ (283.15 K) であり、測定対象は水で満たされた、内寸a [cm] × b [cm] × c [cm]の直方体の水槽である。水槽は厚さ 1 mm のアルミニウム製のものが利用された。先行研究実験で測定された一連の水槽体系について、各水槽の内寸と併せて即発中性子減衰定数αの実験値α_{exp}を表 4.1 に示す。先行研究実験では、水槽体系に対し D-T 中性子源で発生させたパルス中性子を打ち込み、BF3 計数管により中性子計数率の時間変化が測定された。そして、体系内の中性子数が$A\exp(-\alpha t)+C$に従って減衰するとしてデータ点を最小二乗法でフィッティングすることにより即発中性子減衰定数αの実験値が推定された[1]。

表 4.1 過去実験における水槽体系寸法及び実験値α_{exp} [1]

体系番号	a [cm]	b [cm]	c [cm]	α_{exp} [s^{-1}]
1	17.02	17.02	17.73	8012 ± 43
2	17.02	17.02	10.99	9507 ± 56
3	12.07	12.06	12.02	11065 ± 66
4	12.07	12.06	11.88	11083 ± 34
5	10.01	10.03	10.03	13283 ± 115
6	10.01	10.03	8.3	14721 ± 47
7	10.01	10.03	5.99	18228 ± 74
8	6.99	7.12	7	21112 ± 42
9	6.99	7.12	5.73	22897 ± 214
10	6.99	7.12	4.98	25635 ± 68
11	5.71	5.75	5.77	27818 ± 77
12	6.99	7.12	3.96	30551 ± 113
13	5.71	5.75	4.24	33106 ± 98
14	6.99	7.12	3.44	34010 ± 146
15	4.49	4.51	4.4	38838 ± 113
16	4.49	4.51	3.41	44683 ± 182

ここからは、計算に使用した核反応断面積データについて述べる。
まず､計算体系を構成する物質と密度､及び各物質を構成する同位体を表 4.2 に示す｡また、各同位体の質量数及び存在比を表 4.3 に示す。

表 4.2 計算体系を構成する物質の条件

体系温度 T [K]	283.15
水密度 [g/cm^3] [2]	0.9996990014
水の構成同位体	H1, H2, O16, O17, O18
アルミニウム密度［3］	2.702
アルミニウムの構成同位体	Al27

表 4.3 質量数及び同位体存在比 [4]

元素	同位体	質量数	存在比 [atom%]
H	H1	1.00782503223	99.9885
	H2	2.01410177812	0.00115
O	O16	15.99491461957	99.757
	O17	16.99913175650	0.038
	O18	17.99915961286	0.205
Al	Al27	26.981538578	100

ここで、原子数密度は式(4.1)により求めることができる。

$$n = \frac{\rho}{M} \cdot N_A \tag{4.1}$$

n ：原子数密度
ρ ：質量密度
M ：質量数
N_A ：アボガドロ定数

表 4.3 及び式(4.1)から求めた各核種の数密度を表 4.4 に示す。

表 4.4 分子数密度、原子数密度

水分子数密度 [atoms/barn/cm]	$3.34178872283 \times 10^{-02}$
H1 原子数密度 [atoms/barn/cm]	$6.68280883425 \times 10^{-02}$
H2 原子数密度 [atoms/barn/cm]	$7.68611406251 \times 10^{-06}$
O16 原子数密度 [atoms/barn/cm]	$3.33366817623 \times 10^{-02}$
O17 原子数密度 [atoms/barn/cm]	$1.26987971468 \times 10^{-05}$
O18 原子数密度 [atoms/barn/cm]	$6.85066688180 \times 10^{-05}$
Al27 原子数密度 [atoms/barn/cm]	$6.03072515175 \times 10^{-02}$

表 4.4 及び表 4.5 に示す条件のもと、評価済み核データ JENDL-5[5]に対し FRENDY[6],[7],[8]を用いて核データ処理を実施し、計算に使用する中性子エネルギー172 群の巨視的断面積データを求めた。

表 4.5 断面積データ及び処理

評価済み核データ	水 アルミニウム	JENDL-5 update-11 [5]	
評価済み核データ (熱中性子散乱則データ)	H-H2O	JENDL-5 update-11 [5]	T = 280.0 [K]におけるデータを利用
	O-H2O		T = 280.0 [K]におけるデータを利用
	D-D2O		T = 283.6 [K]におけるデータを利用
	アルミニウム		T = 293.6 [K]におけるデータを利用
断面積処理		FRENDY 2.01 [6],[7],[8]	
エネルギー群構造		XMAS 172 群 [9]	

次に、開発した計算コードに与えた計算条件を表 4.6 に示す。なお、この条件は CPU コード、GPU コードともに共通である。

表 4.6 S_N法コードに与える計算条件

計算体系	過去実験水槽体系
空間メッシュ幅	水: 約 0.5 cm (< 0.51 cm)　アルミ: 0.1 cm
境界条件	全面真空境界条件
エネルギー群構造	XMAS 172 群
分点セット	icosahedral quadrature　72 方向
分点の回転	極角・方位角ともに$\pi/5$ rad
非等方散乱の展開次数	3
delta-tracking 法	$\Sigma_{0,g} = \alpha_{\mathrm{est}}/v_g$
収束条件	$\varepsilon_\alpha = 1 \times 10^{-6}$
初期値	均質水体系一点炉近似
浮動小数点数	倍精度
収束加速	詳細メッシュ・詳細群拡散加速
収束加速 適用間隔	詳細計算外部反復 2 回につき 1 回
収束加速	$\varepsilon_{\alpha^{\mathrm{diff}}} = 1 \times 10^{-8}$
収束条件	$\varepsilon_{\phi^{\mathrm{diff}}} = 1 \times 10^{-7}$
浮動小数点数	倍精度

各収束条件におけるεは式(4.2)–(4.4)により計算した。ただし、$|\cdot|$は絶対値、$\left\|\phi^{\mathrm{diff},(n')}\right\|_2$はベクトル$\phi^{\mathrm{diff},(n')}$の L2 ノルムを表す。ただし、添え字diffは拡散加速計算における変数であることを表す。

$$\varepsilon_\alpha = \left| \frac{\alpha^{(n)}}{\alpha^{(n-1)}} - 1 \right| \tag{4.2}$$

$$\varepsilon_{\alpha^{\mathrm{diff}}} = \left| \frac{\alpha^{\mathrm{diff},(n')}}{\alpha^{\mathrm{diff},(n'-1)}} - 1 \right| \tag{4.3}$$

$$\varepsilon_{\phi^{\mathrm{diff}}} = \frac{\left\|\phi^{\mathrm{diff},(n')} - \phi^{\mathrm{diff},(n'-1)}\right\|_2}{\left\|\phi^{\mathrm{diff},(n')}\right\|_2} \tag{4.4}$$

n　：外部反復回数

n'　：拡散加速計算外部反復回数

また、GPU コードについては、3.4.3 項で述べたように version 3 の transport sweep コードを利用し、パラメータは高い性能が得られた$NM_h = 16, NG_h = 2, ND_h = 8$とした。計算

に使用した CPU 及び GPU とその性能を表 4.7 に示す。

表 4.7 使用した CPU 及び GPU とその性能

		理論演算性能 [TFLOPS[5]]		最大メモリ帯域幅 [GB/s]
		単精度	倍精度	
CPU	Intel Core i9-10980XE	2.8	1.4	94
GPU	NVIDIA RTX4090	82.6	1.3	1,008

浮動小数点数演算性能について、使用した GPU は単精度に比べ倍精度の性能が約 1/64 と低いという特徴がある。しかし、第 3 章で述べたように transport sweep の処理のボトルネックは演算ではなく、主にメモリアクセスである。これは非等方散乱中性子源更新や拡散加速計算でも同様である。そのため、本章で実施する GPU による計算においては、倍精度性能の低さの影響は小さいと考えられる。これを踏まえ、計算精度や計算の安定性の観点からも、本章では倍精度浮動小数点数を利用して計算を実施した。

また、GPU では動作周波数が上昇しづらい傾向が見られたため、RTX4090 のベース動作周波数である 2235 MHz に動作周波数を固定して計算を実施した。一方で、CPU についてはそのような傾向は見られなかったため、デフォルト設定のままで実施した。

[5] FLOPS (Floating-point Operations Per Second): 1 秒間に処理可能な浮動小数点数演算の回数

ここで、中性子束の初期値を与える均質水体系一点炉近似について説明する。

本研究で開発したコードでは、中性子束の初期値$\phi_l^{m(0)}$は体系を水均質とした多群一点炉近似に基づき与える。本研究で計算の対象とする体系は、水で満たされたアルミニウム製水槽体系であり、アルミニウム部分は数 cm の体系全体に対して 1 mm と薄い。よって、体系を水均質と近似することで収束解に近い初期値を得られると考えられる。

まず、全中性子束エネルギースペクトルの初期値は、多群一点炉近似を適用した次のα固有値方程式を解くことで、最小固有値に対応する固有ベクトルϕ_gとして得られる[10]。ただし、断面積や拡散係数には、均質としたい材料(本研究の場合は水)に対応するものを与える。また、delta-tracking 法を適用する場合には、適用済みの断面積や拡散係数を用いる。

$$\left(D_g B_g^2 + \Sigma_{\mathrm{t},g}\right)\phi_g - \sum_{g'=1}^{NG} \Sigma_{\mathrm{s0},g'\to g}\phi_g = \frac{\alpha}{\nu_g}\phi_g \tag{4.5}$$

B_g^2 ：エネルギーg群の体系バックリング

ϕ_g ：全中性子束のエネルギースペクトル

真空境界条件下において、体系バックリングB_g^2を次式により与える。このとき、外挿距離として$2.1312D_g$を与えている。これは仮想的に体系表面から外挿距離だけ線形外挿すると全中性子束が零になるとする条件に相当し、外挿距離$2.1312D_g$は輸送理論において等方散乱を仮定した場合に良い近似を与えることが知られている[11]。

$$B_g^2 = \left(\frac{\pi}{L_x + 2\times 2.1312D_g}\right)^2 + \left(\frac{\pi}{L_y + 2\times 2.1312D_g}\right)^2 + \left(\frac{\pi}{L_z + 2\times 2.1312D_g}\right)^2 \tag{4.6}$$

L_x ：x軸方向の体系長さ

L_y ：y軸方向の体系長さ

L_z ：z軸方向の体系長さ

最後に、体系が均質であるとして、得られた全中性子束エネルギースペクトルϕ_gを用いて次式により各メッシュに対して全中性子束の初期値を与える。ただし、$\phi_{0,g,i,j,k}^0$は式(2.46)に基づき規格化する。このとき、$\alpha^{(0)}$には固有値方程式(4.5)を解いて得られる最小固有値を用いる。

$$\phi_{0,g,i,j,k}^0 = \cos(B_x x_i)\cos\left(B_y y_j\right)\cos(B_z z_k)\,\phi_g \tag{4.7}$$

B_x ：式(4.6)第 1 項の正の平方根 $= \pi/(L_x + 2\times 2.1312D_g)$

B_y ：式(4.6)第 2 項の正の平方根 $= \pi/(L_y + 2\times 2.1312D_g)$

B_z ：式(4.6)第 3 項の正の平方根 $= \pi/(L_z + 2\times 2.1312D_g)$

x_i ：x方向i番目メッシュの中心x座標

y_j ：y方向j番目メッシュの中心y座標

z_k ：z方向k番目メッシュの中心z座標

$$\alpha^{(n)} = \frac{1}{\sum_{g=1}^{NG}\sum_{k=1}^{NZ}\sum_{j=1}^{NY}\sum_{i=1}^{NX}\frac{\phi_0^{0(n)}{}_{g,i,j,k}}{v_g}\Delta x_i \Delta y_j \Delta z_k} \quad (2.46)$$

再掲

4.3　α固有値計算コードの妥当性確認及び検証

本節では、先行研究[1]で測定されたα実験値と、開発した CPU 及び GPU コードによるα計算値を比較することで、開発したコードの妥当性確認及び検証を実施する。また、4.3.2 項では熱中性子散乱則に起因するα計算値の不確かさ評価を実施する。

4.3.1　妥当性確認及び GPU コードの検証

各体系におけるαの実験値α_{exp}と、4.2 項の条件のもと CPU, GPU コードにより得られた計算値$\alpha_{\mathrm{calc,CPU}}$，$\alpha_{\mathrm{calc,GPU}}$を表 4.8 に示す。また、横軸を$\alpha_{\mathrm{exp}}$、縦軸を$\alpha_{\mathrm{calc}}$としたプロットを図 4.2 に示す。

表 4.8 実験値α_{exp}と計算値α_{calc}の比較

体系番号	α_{exp} [s^{-1}]	$\alpha_{\mathrm{calc,CPU}}$ [s^{-1}]	$\alpha_{\mathrm{calc,GPU}}$ [s^{-1}]	$\alpha_{\mathrm{calc,CPU}} - \alpha_{\mathrm{calc,GPU}}$ 相対差異 [-]
1	8012 ± 43	8122	8122	-2.9×10^{-7}
2	9507 ± 56	9628	9628	-2.9×10^{-7}
3	11065 ± 66	11251	11251	-2.7×10^{-7}
4	11083 ± 34	11298	11298	-2.7×10^{-7}
5	13283 ± 115	13835	13835	-1.7×10^{-7}
6	14721 ± 47	15067	15067	-1.7×10^{-7}
7	18228 ± 74	18431	18431	-1.2×10^{-7}
8	21112 ± 42	21806	21806	-1.1×10^{-7}
9	22897 ± 214	24200	24200	-1.0×10^{-7}
10	25635 ± 68	26424	26424	-7.3×10^{-8}
11	27818 ± 77	28930	28930	-9.5×10^{-8}
12	30551 ± 113	31312	31312	-9.5×10^{-8}
13	33106 ± 98	34345	34345	-8.7×10^{-8}
14	34010 ± 146	35287	35287	-6.4×10^{-8}
15	38838 ± 113	41321	41321	-4.7×10^{-8}
16	44683 ± 182	47459	47459	-3.2×10^{-8}

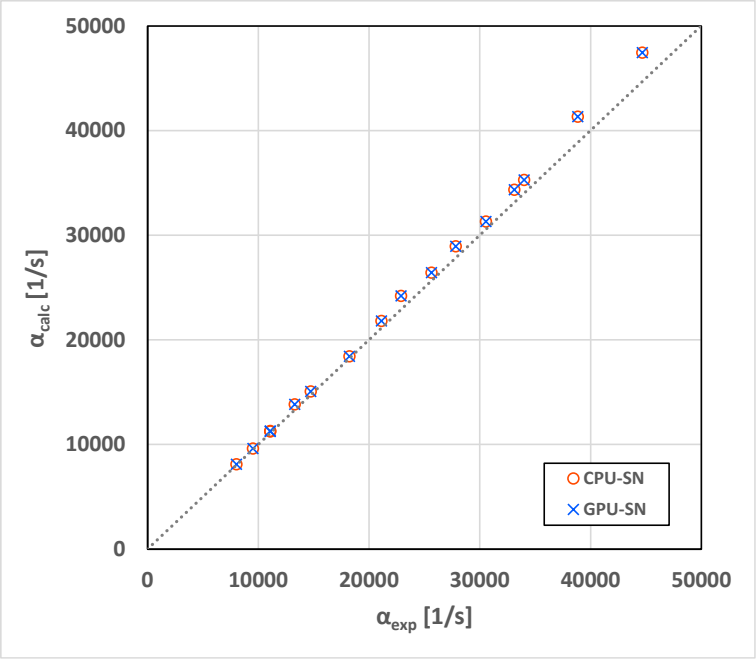


図 4.2 実験値α_{exp}と計算値α_{calc}の比較 (破線は$\alpha_{exp} = \alpha_{calc}$の直線)

加えて、実験値α_{exp}と計算値α_{calc}の相対差異$(\alpha_{calc}/\alpha_{exp} - 1)$を図 4.3 に示す。ただし、図中のエラーバーは実験値不確かさとして、先行研究[1]に示されたαのフィッティング誤差に起因する誤差を示している。

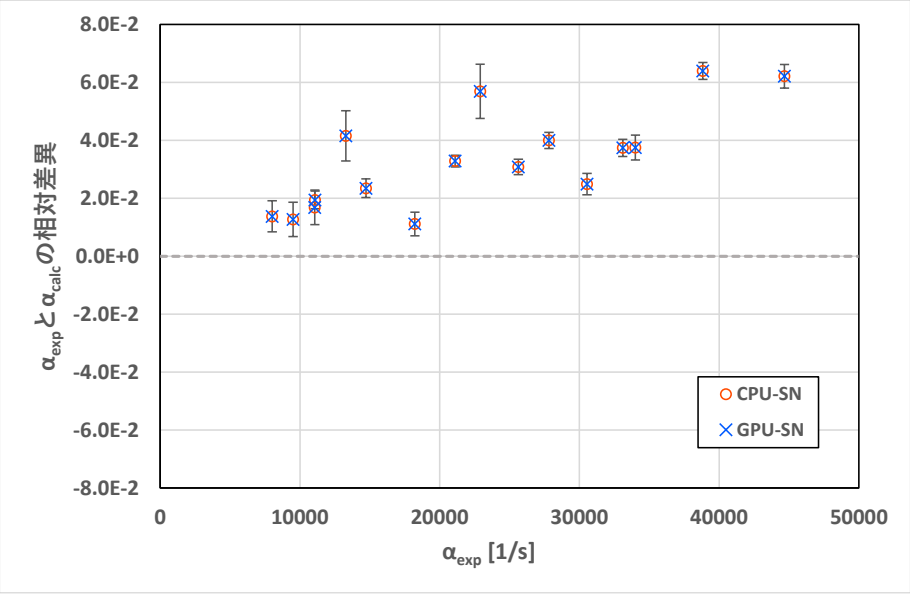


図 4.3 実験値α_{exp}と計算値α_{calc}の相対差異

表 4.8、図 4.2 及び図 4.3 から、実験値α_{exp}と計算値α_{calc}はおおむね一致しており、開発したコードの妥当性を確認できた。また、CPU コードと GPU コードによる計算値の相対差異は全ての計算で設定した収束条件$\varepsilon_{\alpha} = 1 \times 10^{-8}$に近い値である$3 \times 10^{-7}$を下回っていることから、GPU コードは CPU コードとほぼ同等の計算が可能であり、開発した GPU コードが正しく実装されていることを確認できた。

ただし、CPU コードと GPU コードいずれの結果でも、実験値不確かさの範囲内に実験値と計算値の差異は収まっておらず、実験値α_{exp}が大きくなるにつれて実験値に対する計算値の過大評価が徐々に大きくなることが明らかとなった。この原因として、①S_N法の手法起因のα不確かさや、②使用した評価済み断面積不確かさに起因した不確かさ、あるいは③先行研究における実験値不確かさの過小評価、などの原因が可能性として考えられる。

以降では、断面積起因不確かさについて考察する。まず、計算体系は水とわずかなアルミニウムのみからなるため、中性子エネルギーは主に熱中性子領域に分布した形状となっている。また、αが大きい体系は中性子の漏洩量が大きい体系に相当することから、水槽の寸法が小さくなるにつれて、水の散乱断面積がα計算値に与える影響がより大きくなると考えられる。ゆえに、αの増大に伴い計算値が徐々に過大評価される原因は、熱中性子に対する散乱断面積の不確かさに起因するものであると考えられる。

なお、先行研究[10]の検討結果によると、核データ共分散(SCALE コードシステム付属の 252 群共分散データ scale.rev08.252groupcov7.1 [3])に起因したα不確かさは、実験値不確かさの約半分と推定されている。ただし、先行研究[10]の不確かさ評価結果には、水の熱中性子散乱則(TSL, Thermal Scattering Law)による寄与は含まれていない。従って、水槽体系におけるα実験値とα計算値の差異について議論するためには、水の熱群散乱断面積に大きく影響を及ぼす、^{1}H の熱中性子散乱則の不確かさがα計算値に与える影響を評価する必要があると考えた。そこで次項では、^{1}H の熱中性子散乱則に起因するα計算値の不確かさを評価する。

4.3.2　熱中性子散乱則に起因する不確かさの評価

本項では、^{1}H 熱中性子散乱則の不確かさに起因するα計算値の不確かさを評価する。

^{1}H 熱中性子散乱則起因の不確かさを評価するため、unscented 変換に基づく決定論的サンプリング[12]によりサンプリングされた 13 個の HinH2O TSL データを用いた[13],[14]。

摂動対象のパラメータは CAB モデルに基づいた ^{1}H 熱中性子散乱則[15],[16],[17]の 6 つのパラメータであり、これらのパラメータを決定論的サンプリングにより独立に摂動させた 13 個の LEAPR ファイルを生成し、これらのファイルを NJOY2016[18]により 13 個の摂動された TSL データを得た。摂動対象のパラメータと相対標準偏差を表 4.9 に示す。unscented 変換に基づく決定論的サンプリングでは、未摂動の 6 つのパラメータについては ENDF/B-VIII.0[19]の値を用いた上で、13 個の標本重みがすべて等しく($w_i = 1/13$)なるように、6 つのパラメータの摂動量を設定した。

なお、HinH2O TSL データ以外の条件は 4.2 項で述べた条件と同様であり、α固有値計算は GPU コードによってのみ実施した。

表 4.9 摂動対象パラメータと相対標準偏差

パラメータ	相対標準偏差
Δ (scaling factor)	$\pm 10\%$
σ_s (elastic cross section)	$\pm 0.2\%$
ω_t (translational weight)	$\pm 15\%$
E_1 (first oscillator energy)	$\pm 5\%$
E_2 (second oscillator energy)	$\pm 30\%$
c (diffusion coefficient)	$\pm 0.5\%$

以上で述べた 13 個の TSL データに基づいて推定した不確かさをエラーバーとして図 4.2 に追加した結果を図 4.4 に示す。図 4.4 より、いずれの場合においても実験値と計算値の差異は、水の TSL データ起因不確かさ2σの範囲内に収まっていることが確認できた。このことから、水槽体系では水の TSL データがα計算値に大きな影響を及ぼしており、α計算値の過大評価の原因は主に TSL データ不確かさに起因していると考えられる。

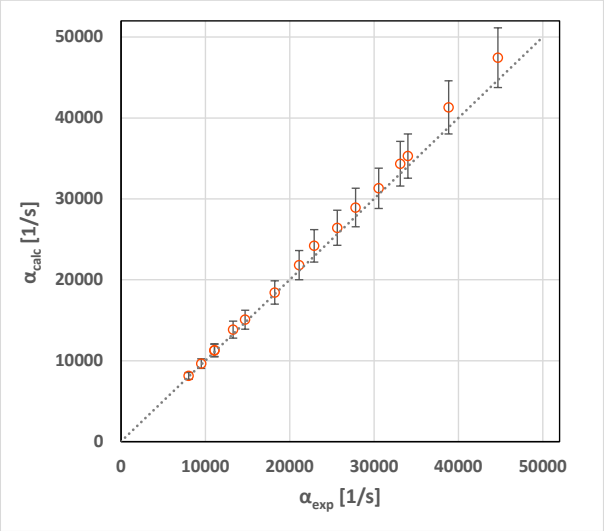


図 4.4 実験値α_{exp}と ^{1}H TSL 起因不確かさを考慮した計算値$\alpha_{calc,GPU}$の比較
(エラーバーは2σの範囲を示している)

4.4 GPGPU を用いたα固有値計算コードの有効性検証

本節では、開発した GPU コードの有効性を計算時間の観点から検証する。4.4.1 項では、CPU コード及び GPU コードによるα固有値計算を実施し、計算の収束までに要する時間を測定、比較することで、GPGPU 活用の有効性を検証する。4.4.2 項では、4.4.1 項の結果を受けて、どのような条件下で GPGPU の利用が有効的であるのかを確認し、その理由について考察する。

4.4.1 CPU コード・GPU コードによる計算時間の比較

表 4.10 に全計算体系における、CPU・GPU コードによるα固有値計算収束に要した時間をそれぞれ示す。計測は C++標準ライブラリに含まれる `std::chrono::system_clock` クラスによる。測定対象には一点炉近似による初期値設定や host・device 間のメモリ転送時間等を含む。また、CPU コードの散乱源更新計算と transport sweep はループ変数gについて 18 thread に並列化している。

表 4.10 CPU・GPU コードにおけるα固有値計算収束までの所要時間の比較

体系番号	空間メッシュ分割数 $NX \times NY \times NZ$	T_{CPU} [s]	T_{GPU} [s]	高速化率
1	$36 \times 36 \times 36$	2159.9	196.8	11.0
2	$36 \times 36 \times 23$	1075.7	110.4	9.7
3	$26 \times 26 \times 25$	487.4	62.7	7.8
4	$26 \times 26 \times 25$	485.4	61.9	7.8
5	$22 \times 22 \times 21$	256.1	36.9	6.9
6	$22 \times 22 \times 18$	197.8	31.7	6.2
7	$22 \times 22 \times 13$	162.0	27.4	5.9
8	$16 \times 16 \times 15$	94.8	17.3	5.5
9	$16 \times 16 \times 13$	95.1	18.0	5.3
10	$16 \times 16 \times 11$	84.4	15.8	5.3
11	$14 \times 14 \times 13$	60.5	14.9	4.1
12	$16 \times 16 \times 9$	64.6	13.3	4.9
13	$14 \times 14 \times 10$	44.4	12.2	3.6
14	$16 \times 16 \times 8$	68.2	10.3	6.6
15	$12 \times 12 \times 11$	32.1	10.5	3.1
16	$12 \times 12 \times 11$	33.7	11.2	3.0

表 4.10 より、GPGPU の利用によって、最も寸法が大きな(空間メッシュ分割数が多い)水槽体系について、最大で 11.0 倍の高速化を達成できたことを確認した。一方で、体系が小

さくなるほど高速化率は低下し、小さな体系では高速化率が 3.0 倍まで低下している。次項ではこの原因を考察する。

4.4.2　考察

体系が小さくなるほど高速化率が低下する原因を確かめるため、体系番号 1, 16 のα固有値計算における各処理に要した時間をそれぞれ図 4.5、図 4.6 に示す。図中で左側のグラフが CPU コード、右側のグラフが GPU コードによる計算時間を示している。ただし、4.1 項で示したように拡散加速計算は外部反復奇数回目にのみ適用していることに注意されたい。また、GPU コードについては host・device 間のデータ転送に要した時間を併せて示している。

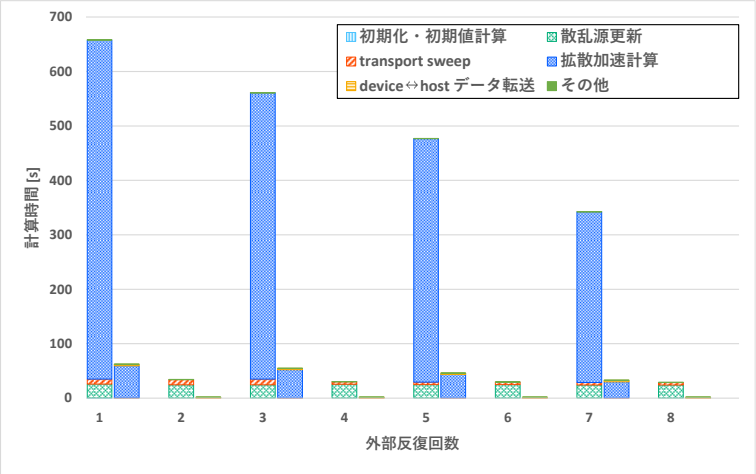


図 4.5 体系番号 1 CPU・GPU コードで各計算に要した時間の比較
左：CPU コード　右：GPU コード

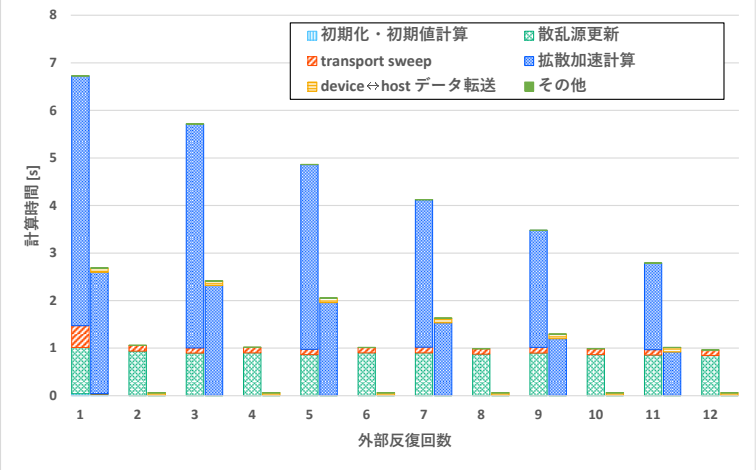


図 4.6 体系番号 16 CPU・GPU コードで各計算に要した時間の比較
左：CPU コード　右：GPU コード

加えて、α固有値計算における拡散加速計算の処理時間と GPU コードによる高速化率を表 4.11 に示す。

表 4.11 計算体系 1, 16 における拡散加速計算の処理時間と GPU コードによる高速化率 (1 回の処理あたりの平均値)

	体系番号 1		体系番号 16	
	処理時間 [s]	高速化率	処理時間 [s]	高速化率
CPU	477.30	10.4	3.54	2.0
GPU	45.96		1.74	

図 4.5、図 4.6 で整理した結果より、いずれの計算体系においても拡散加速計算に要した時間が最も多くを占めており、特に GPU コードでは総計算時間の 9 割以上を占めるボトルネックとなっていることが分かる。また表 4.11 から、体系番号 16 では拡散加速計算の GPU による高速化率が体系番号 1 の 8.6 倍から 1.6 倍まで悪化している。

ここで、拡散加速計算で発行される thread の数について考えると、体系番号 1 の空間メッシュ数が 46,656 であるのに対して体系番号 16 の空間メッシュ数は 1,584 であり、その分発行できる thread は少なくなる。MAGMA[20],[21]や、MAGMA が依存する線形代数ライブラリ cuBLAS[22]及び疎行列ライブラリ cuSPARSE[23]関数の実行に際して発行される thread 数は明らかではないが、おおむね$NX \times NY \times NZ \times NG$に比例する数の thread が発行されると考えられる。よって、体系番号 16 のような小さな体系、すなわち空間メッシュ数が少ない体系で拡散加速計算の高速化率が低下する原因は、拡散加速計算の計算時間のほとんどを占める BiCGSTAB 法[24]の計算(疎行列計算)において、GPU の性能を使い切るのに十分な数の thread を発行できないことにあると考えられる。

また、拡散加速計算の占める計算時間が全体のボトルネックとなる点については、拡散加速計算にさらに加速を施すことで改善できると考えられる。本コードでは拡散加速計算の収束に数十～数百回ほどの外部反復を要しているものの、各回の計算は数ミリ秒～数百ミリ秒と十分に高速である。ゆえに、拡散加速計算に対して空間均質化あるいはエネルギー群縮約を施した CMFD 加速法[25]を適用することで外部反復回数を削減し、より高速な拡散加速計算を実現できると考えられる。ただし、空間均質化やエネルギー群縮約に伴い必要となる計算が各空間メッシュ・各エネルギー群に対して生じるため、これらの計算に GPGPU を活用するなどして、ボトルネックとならないよう十分高速に実施する必要があることに注意しなければならない。

次に、図 4.5 及び図 4.6 で整理した結果から、拡散加速計算の計算時間を除いて整理しなおした図をそれぞれ図 4.7 及び図 4.8 に示す。

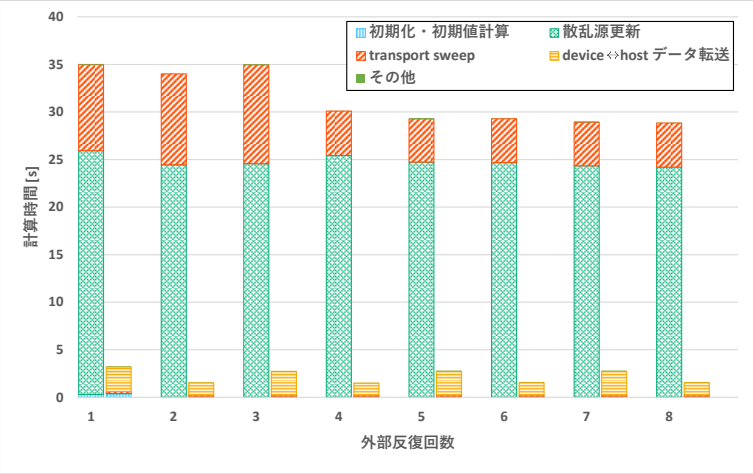


図 4.7 体系番号 1 CPU・GPU コードで各計算に要した時間の比較 (拡散加速計算を除く)
左：CPU コード　右：GPU コード

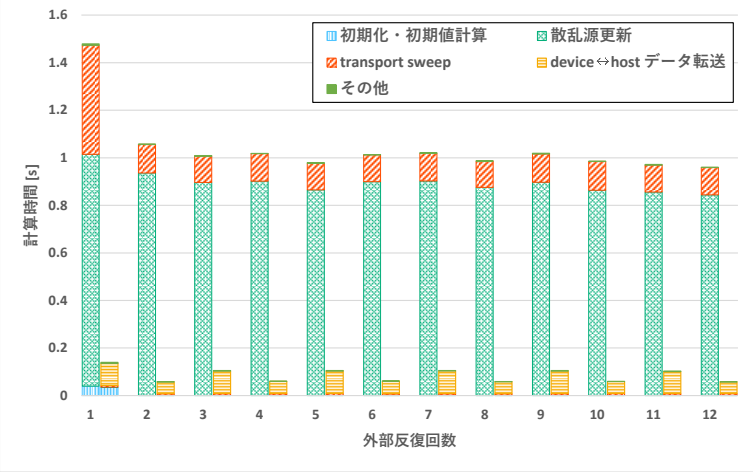


図 4.8 体系番号 16 CPU・GPU コードで各計算に要した時間の比較 (拡散加速計算を除く)
左：CPU コード　右：GPU コード

また、計算体系 1、16 における散乱源更新と transport sweep について、処理時間と GPU による高速化率を表 4.12 に示す。加えて、計算体系 16 に対する計算体系 1 のメッシュ分割数と GPU による処理時間の比を表 4.13 に示す。

表 4.12 計算体系 1、16 における散乱源更新及び transport sweep の処理時間と高速化率 (詳細計算外部反復 1 回あたりの平均値)

		体系番号 1		体系番号 16	
		処理時間 [s]	高速化率	処理時間 [s]	高速化率
散乱源更新	CPU	24.7546	540	0.8928	471
	GPU	0.0459		0.0019	
transport sweep	CPU	6.5073	34.1	0.1436	21.3
	GPU	0.1908		0.0068	
host・device 間データ転送	GPU	1.9073	-	0.0720	-

表 4.13 計算体系 16 に対する計算体系 1 のメッシュ分割数と処理時間の比 (詳細計算外部反復 1 回あたりの平均値)

		体系番号 1	体系番号 16	#1 / #16
メッシュ分割数		46,656	1,584	29.5
CPU 処理時間 [s]	散乱源更新	24.7546	0.8928	27.7
	transport sweep	6.5073	0.1436	45.3
GPU 処理時間 [s]	散乱源更新	0.0459	0.0019	24.2
	transport sweep	0.1908	0.0068	28.3

表 4.12 より、最も寸法が大きな水槽体系について、散乱源更新では約 540 倍、transport sweep では約 34.1 倍の高速化を達成できたことを確認した。

また、表 4.13 より、GPU による散乱源更新及び transport sweep の処理時間は、メッシュ分割数におおむね比例することが分かる。一方で transport sweep の GPU による高速化率については、表 4.12 に示したように、計算体系 1 では 34.1 倍、計算体系 16 では 21.3 倍と、約 1.6 倍の差がある。これは、CPU による transport sweep の処理時間がメッシュ分割数に比例していないことによる。本研究では、その原因について詳しく考察できていないものの、メモリアクセスパターンの変化によるキャッシュヒット率の変化などが原因として考えられる。

計算が高速化できた一方で、図 4.7、図 4.8、表 4.12 からわかるように、transport sweep の処理時間に比べて host・device 間のデータ転送時間が約 10 倍の時間の要している。計算全体では拡散加速計算がボトルネックであるため、データ転送時間による影響は小さいと

いえる。しかし、拡散加速計算をさらに高速化し他の処理と処理時間が同程度のオーダーにできた場合には、データ転送はボトルネックとなり得る。

開発した GPU コードでは、拡散加速計算における中性子流補正係数D_{cor}や係数行列$\mathbf{A}$の計算をはじめとして、host(CPU)側により処理を実施する箇所が複数あり、それに伴い host・device 間で全中性子束や中性子流などのデータの転送を行っている。3.2.1 項で述べたように、host・device 間のデータ転送は大きなオーバーヘッドを伴うため、多数回の小さなデータ転送を行うことは避けるべきである。しかし、開発しコードでは考慮できておらず、データ転送に長時間を要している。これを改善するには、host 側による処理をしている個所を device 側により処理できるようにすることで、データ転送の量や回数を減らす必要がある。具体的には、拡散加速計算における中性子流補正係数D_{cor}や係数行列$\mathbf{A}$の計算を device 側で処理できるようにする必要がある。

ここまでで示した結果は、飛行方向分割数$ND = 72$、展開次数上限$NL = 3$の場合の結果であり、この条件下では拡散加速計算がボトルネックとなっていた。一方で、散乱源更新計算の計算量はNLに、transport sweep の計算量はNDとNLに依存して増加するため、ND, NLが大きな計算条件下ではこれらの計算がより多くの計算時間を占めるようになると考えられる。そこで体系番号 1 の計算について、ND, NLを大きくした$ND = 1200, NL = 7$の場合における計算時間を測定した。測定結果を図 4.9 及び表 4.14 に示す。

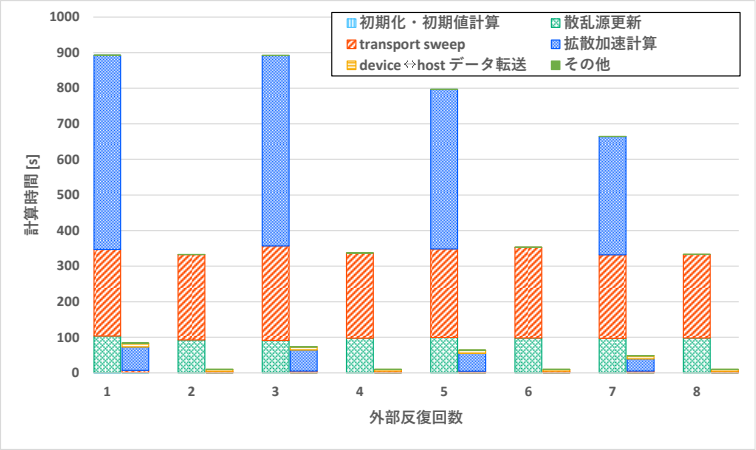


図 4.9 CPU・GPU コードで各計算に要した時間の比較(計算体系 1、$ND = 1200$、$NL = 7$)
左：CPU コード　右：GPU コード

表 4.14 計算体系 1、各条件下における各処理の GPU コードによる高速化率

	GPU コードによる高速化率	
	$ND = 72, NL = 3$	$ND = 1200, NL = 7$
散乱源更新	540	531
transport sweep	34.1	49.6
拡散加速計算	10.4	8.9
計	10.7	14.9

図 4.9 より、CPU コードを用いた場合、$ND = 1200, NL = 7$の条件下では transport sweep 及び散乱源更新に要する時間がより多くを占めていることがわかる。これにより、拡散加速計算による律速の影響が小さくなり、表 4.14 で示すように、全体の高速化率が 10.7 倍から 14.9 倍まで向上した。

また、transport sweep の GPU による高速化率が$ND = 72, NL = 3$の場合に比べて向上しており、これは主に発行する block 数が増えたことに起因すると考えられる。

kernel を飛行方向の象限ごとに実行することを踏まえると、$ND = 72, NL = 3$の場合の発

行する block 数はceil$(NG, NG_h)/NG_h \times$ ceil$(ND/8, ND_h)/ND_h =$ ceil$(172,2)/2 \times$ ceil$(72/8,8)/8 = 86 \times 2 = 172$ block 程度となる。$ND = 1200, NL = 7$ の場合は、ceil$(172,2)/2 \times$ ceil$(1200/8,8)/8 = 86 \times 20 = 1720$block 程度となる。ただし、この block 数の推定値は icosahedral 分点に回転処理を施した後の各象限に属する飛行方向の数により変動する。

$ND = 72, NL = 3$の場合の 172 block は GPU の搭載する SM 128 基に対して十分な数ではないため、各 SM に対する負荷が不均一になりやすく、occupancy も低下しやすいため、十分な性能を発揮できていなかったと考えられる。

ここで、3.4.3 項で用いたテスト体系について、$(ND, NL) = (72, 3), (1200, 7)$のそれぞれの場合で性能を測定した結果の一部を図 4.10 に示す。図 4.10 より、$(ND, NL) = (72, 3)$の場合は block が少ないため、theoretical occupancy が 66.67%であるのに対して achieved occupancy は 22.57%と低く、それに伴って Eligible Warps Per Scheduler の値も低くなっていることがわかる。このことは、割り当てられた block の数が十分でなく、占有率が低下した SM が多数存在することを示唆している。

なお、NLが大きい、すなわち不可分操作が多数回要求される条件下であっても高い性能を発揮できていることから、不可分操作による性能への影響は小さく抑えられているといえる。

以上のことから、開発した GPU コードは$(ND, NL) = (72, 3)$のような場合でも CPU コードに比べ高速ではあるが、$(ND, NL) = (1200, 7)$のようにND, NLが十分大きい場合により性能を発揮できることを明らかにした。

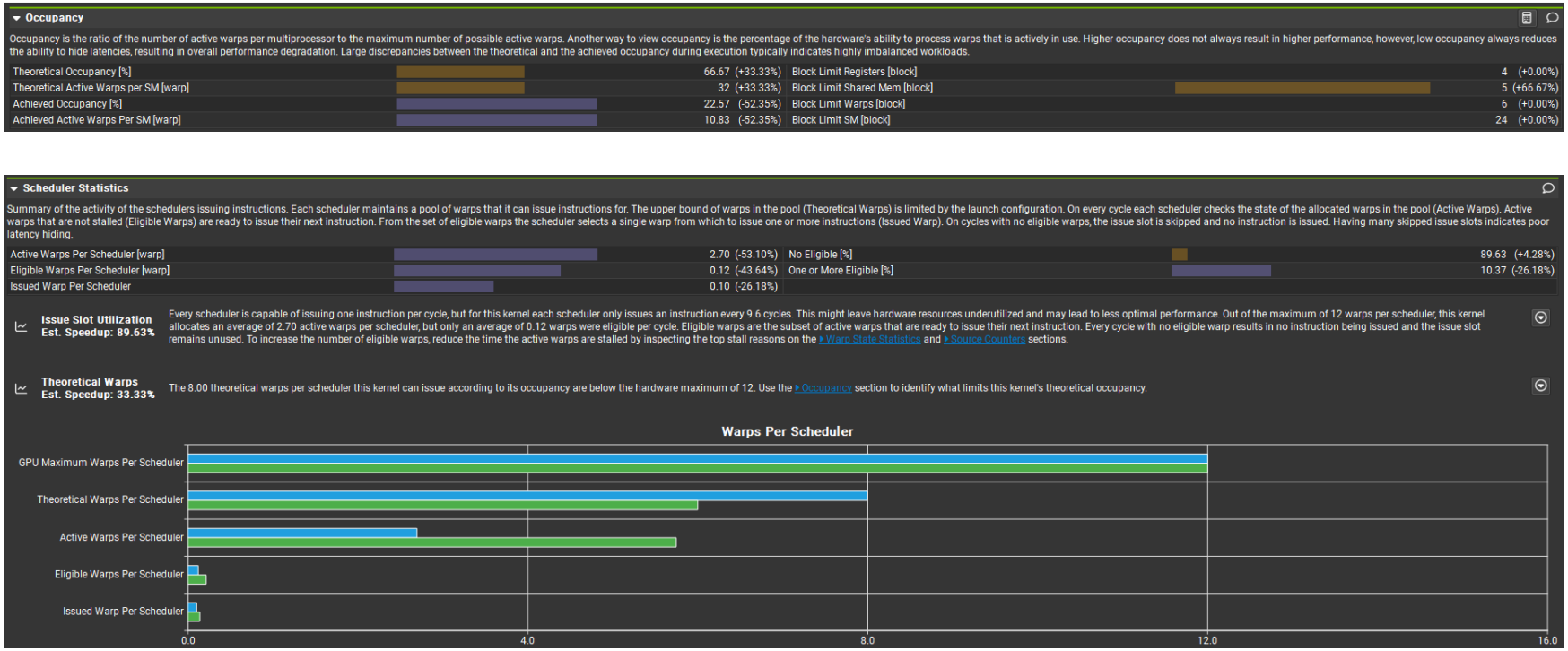


図 4.10 テスト体系における$(ND, NL) = (1200, 7)$のそれぞれの場合での性能測定結果

括弧内の百分率は$(ND, NL) = (1200, 7)$のときを基準とした$(ND, NL) = (72, 3)$のときの性能

青：$(ND, NL) = (72, 3)$　　緑：$(ND, NL) = (1200, 7)$

NVIDIA Nsight Compute による測定 繰り返し回数：5 回

4.5　本章のまとめ

本章では、第 2 章及び第 3 章に基づき開発した、GPGPU を用いたS_N法に基づくα固有値計算コードの妥当性や高速化の効果を検証した。

4.2 節では、本章の妥当性確認及び検証計算で、過去実験[1]の水槽体系を計算の対象とすることと、計算コードに与えた計算条件について述べた。エネルギー群構造は XMAS 172 群、分点セットには icosahedral 分点を利用し、飛行方向分割数NDは72とした。また、非等方散乱の展開次数NLは 3 次とした。

4.3 節では、先行研究によるα実験値と、開発した CPU 及び GPU コードによるαの計算結果を比較することで、開発したコードの妥当性確認及び検証を実施した。4.3.1 項では、CPU、GPU コードそれぞれによるα計算値$\alpha_{\mathrm{calc,CPU}}$と$\alpha_{\mathrm{calc,GPU}}$が収束条件と同程度の範囲で一致していることから、GPU コードが正しく実装できていることを確認したとともに、α_{exp}とα_{calc}がよく一致することからコードの妥当性を確認した。一方で、水槽体系が小さくなるにつれα_{exp}とα_{calc}の差異が大きくなっており、その原因を考察するため、熱中性子散乱則に起因するα計算値の不確かさを決定論的サンプリング法に基づき推定した。結果、α_{exp}とα_{calc}の差異は H in H2O TSL データに起因する不確かさ2σの範囲内であり、TSL データがα_{calc}に大きな影響を与えている可能性があることが分かった。

4.4 節では、開発した CPU 及び GPU コードの計算時間を比較し、GPU コードの有効性を計算時間の観点から検証した。$ND = 72, NL = 3$の条件下で、最も寸法が大きな水槽体系について最大で 11.0 倍の高速化を達成できたことを確認した。一方で、どの計算体系においても拡散加速計算に要する時間が 6 割から 9 割を占めており、GPU による拡散加速計算の高速化率が低下する小さな寸法の体系においては、拡散加速計算に律速され全体の高速化率も低くなることが分かった。そこで、飛行方向分割数NDと非等方散乱の展開次数NLを大きくした$ND = 1200, NL = 7$の条件下で計算を実施し、性能を測定した。その結果、transport sweep と非等方散乱源更新に要する時間が増加したことで拡散加速計算によるボトルネックの影響が低減され、全体の高速化率が 14.9 倍まで向上した。また、発行できる block 数が増加したことで、GPU による transport sweep の性能が向上したことも確認した。

本章では、開発しコードが正しく実装できていることと、コードの妥当性を確認することができた。また、条件によっては計算全体で 14.9 倍の高速化を達成できたことから、開発したコードの有効性についても確認することができた。一方で、GPU コードにおいて計算時間の多くを占める拡散加速計算のさらなる高速化が今後の課題として挙げられる。

4.6 参考文献

[1] K. KOBAYASHI, Y. SEKI, N. MIZOO, et al., “Measurement and Calculation of Neutron Diffusion Parameters in Water,” Journal of Nuclear Science and Technology 3 7, 275 (1966); https://doi.org/10.1080/18811248.1966.9732325.

[2] F. E. JONES and G. L. HARRIS, “ITS-90 density of water formulation for volumetric standards calibration,” J. RES. NATL. INST. STAN. 97 3, 335 (1992); https://doi.org/10.6028/jres.097.013.

[3] W. A. WIESELQUIST, R. A. LEFEBVRE, and M. A. JESSEE, “SCALE Code System,” ORNL/TM-2005/39 Version 6.2.4, 1616812, p. ORNL/TM-2005/39 Version 6.2.4, 1616812 (2020); https://doi.org/10.2172/1616812.

[4] “Tables of Nuclear Data;” JAEA, https://wwwndc.jaea.go.jp/NuC/index_J.html; (current as of Jan. 3, 2024).

[5] O. IWAMOTO, N. IWAMOTO, S. KUNIEDA, et al., “Japanese evaluated nuclear data library version 5: JENDL-5,” Journal of Nuclear Science and Technology 60 1, 1 (2023); https://doi.org/10.1080/00223131.2022.2141903.

[6] K. TADA, A. YAMAMOTO, S. KUNIEDA, et al., “Development of nuclear data processing code FRENDY version 2,” Journal of Nuclear Science and Technology, 1 (2023); https://doi.org/10.1080/00223131.2023.2278600.

[7] A. YAMAMOTO, K. TADA, G. CHIBA, et al., “Multi-group neutron cross section generation capability for FRENDY nuclear data processing code,” Journal of Nuclear Science and Technology 58 11, 1165 (2021); https://doi.org/10.1080/00223131.2021.1921631.

[8] K. TADA, R. KONDO, T. ENDO, et al., “Development of ACE file perturbation tool using FRENDY,” Journal of Nuclear Science and Technology 60 6, 624 (2023); https://doi.org/10.1080/00223131.2022.2130463.

[9] E. SARTORI and G. C. PANINI, “OECD/NEA Data Bank: Standard Energy Group Structures of Cross Section Libraries for Reactor Shielding, Reactor Cell and Fusion Neutronics Applications: VITAMIN-J, ECCO-33, ECCO-2000 and XMAS JEF/DOC-315 Revision 3 - DRAFT,” (1991).

[10] T. ENDO, A. NOGUCHI, A. YAMAMOTO, et al., “Perturbation-Theory-Based Sensitivity Analysis of Prompt Neutron Decay Constant for Water-Only System”, *Transactions of American Nuclear Society*, 124, pp.184-187 (2021).

[11] K. KOBAYASHI, *原子炉物理*, コロナ社, Tokyo (1996).

[12] Y. FUKUI, T. ENDO, and A. YAMAMOTO, “Nuclear data adjustment using a deterministic sampling method with unscented transformation,” Journal of Nuclear Science and Technology 60 3, 238 (2023); https://doi.org/10.1080/00223131.2022.2095051.

[13] Y. HARADA, H. YAMAGUCHI, T. ENDO, et al., “Uncertainty Quantification of Prompt Neutron Decay Constant α due to the Thermal Neutron Scattering Law of Water,” M&C 2023, Ontario, Canada, Aug. 13–17, 2023, American Nuclear Society (2023).

[14] Y. HRADA, H. YAMAGUCHI, T. ENDO, et al., “即発中性子減衰定数を用いたデータ同化による軽水の熱中性子散乱則に起因した不確かさの低減,” 日本原子力学会 2024 春の年会, Higashiosaka, Japan, Mar. 26–28, 2024.

[15] J. I. MÁRQUEZ DAMIÁN, J. R. GRANADA, and D. C. MALASPINA, “CAB models for water: A new evaluation of the thermal neutron scattering laws for light and heavy water in ENDF-6 format,” Annals of Nuclear Energy 65, 280 (2014); https://doi.org/10.1016/j.anucene.2013.11.014.

[16] “ TENDL-2021 Thermal scattering data,” TENDL-2021, https://tendl.web.psi.ch/tendl_2021/randomTSL.html.

[17] D. ROCHMAN, A. VASILIEV, H. FERROUKHI, et al., “Impact of H in H 2 O thermal scattering data on criticality calculation: uncertainty and adjustment,” EPJ Nuclear Sci. Technol. 8, 3 (2022); https://doi.org/10.1051/epjn/2021028.

[18] R. MACFARLANE, D. W. MUIR, R. M. BOICOURT, et al., “The NJOY Nuclear Data Processing System, Version 2016,” LA-UR--17-20093, 1338791, p. LA-UR--17-20093, 1338791 (2017); https://doi.org/10.2172/1338791.

[19] D. A. BROWN, M. B. CHADWICK, R. CAPOTE, et al., “ENDF/B-VIII.0: The 8th Major Release of the Nuclear Reaction Data Library with CIELO-project Cross Sections, New Standards and Thermal Scattering Data,” Nuclear Data Sheets 148, 1 (2018); https://doi.org/10.1016/j.nds.2018.02.001.

[20] “MAGMA;” Innovative Computing Laboratory; https://icl.utk.edu/magma/; (current as of Jan. 15, 2024).

[21] H. ANZT, W. Sawyer, S. TOMOV, et al., “Optimizing Krylov Subspace Solvers on Graphics Processing Units,” in 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, pp. 941–949, IEEE, Phoenix, AZ, USA (2014); https://doi.org/10.1109/IPDPSW.2014.107.

[22] “cuBLAS,” NVIDIA Corporation; https://developer.nvidia.com/cublas; (current as of Jan. 15, 2024).

[23] “cuSPARSE,” NVIDIA Corporation; https://developer.nvidia.com/cusparse; (current as of Jan. 24, 2024).

[24] H. A. VAN DER VORST, “Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems,” SIAM J. Sci. and Stat. Comput. 13 2, 631 (1992); https://doi.org/10.1137/0913035.

[25] N. Z. CHO, C. J. PARK, “A Comparison of Coarse Mesh Rebalance (CMR) and Coarse Mesh Finite Difference (CMFD) Acceleration Methods for the Neutron Transport Calculations,” Proc. M&C 2003, Gatlinburg, Tennessee, April 6–11, 2003, American Nuclear Society (2003);

第5章　結論

5.1　まとめ

本研究では、GPGPU を用いた決定論的手法に基づくα固有値計算の高速化に関する検討を行った。各章のまとめを以下に述べる。

第 1 章では、本研究の背景と目的について述べた。

軽水炉の詳細設計などに使用される軽水の中性子散乱断面積データは、幅広い中性子エネルギーと媒質温度範囲において信頼性が必要であり、特に熱中性子は軽水炉の核分裂反応に大きく寄与するため熱中性子散乱則(TSL)データが重要とされる。しかし、臨界実験結果を用いたデータ同化では核燃料に含まれる ^{235}U などの核種がk_{eff}に影響を与えることから軽水の TSL を選択的に改善することは困難であった。そこで、水槽体系のような未臨界体系でも測定可能な即発中性子減衰定数αの測定結果と数値解析結果を活用することで、選択的な TSL の改善が可能になると期待されている。

一方で、典型的な軽水炉を対象としたk_{eff}固有値計算では、非等方散乱中性子源をP_3成分まで取り扱うことが多い。同様に軽水を多量に含む水槽体系においても、核特性を精度良く評価するためには非等方散乱中性子源を正確に取り扱う必要がある。また先行研究において、中性子飛行方向を考慮可能なS_N法基づくα固有値計算コードが試作されたものの、多数回の反復輸送計算が必要であり大きな計算コストを伴うことが課題となっている。

そこで本研究では、GPGPU と呼ばれる GPU を計算の高速化などの画像処理以外の目的で利用する技術に注目し、GPGPU を活用してS_N法に基づくα固有値計算を高速化することを目的とした。

第 2 章では、S_N法に基づくα固有値計算理論について述べた。

まず、S_N法に基づきエネルギー、飛行方向、空間について離散化及び差分化したα固有値方程式の導出を示した。k_{eff}固有値計算では核分裂源項$\nu\Sigma_{\mathrm{f}}\psi/k_{\mathrm{eff}}$が固有値方程式の右辺に現れるが、$\alpha$固有値計算では右辺に$(\alpha/v)\psi$を加える代わりに、左辺の巨視的全断面積$\Sigma_{\mathrm{t}}$を$(\Sigma_{\mathrm{t}}-\alpha/v)$と補正することで計算を行う。また、実球面調和関数展開を用いた非等方散乱中性子源の取り扱いと、少ない飛行方向分割数で高精度な球面調和関数の積分計算が可能である特徴を持つ icosahedral 分点について説明した。そして、反復法によるα固有値の数値解法について述べた。

次に、S_N法の収束を加速するための拡散加速法について述べた。拡散加速法の基本となる拡散理論に基づくα固有値方程式を示すとともに、詳細計算の正味の中性子流を再現するための中性子流補正係数とその導出を示した。このとき、詳細計算(S_N法計算)において巨視的全断面積を$(\Sigma_{\mathrm{t}}-\alpha/v)$と補正するような計算をしていることから、それに合わせて拡散係数も補正することとした。そして、解くべきα固有値方程式と反復法によるα固有値の数値解

法について述べた。

また、アルミニウムなどの巨視的全断面積の小さい材質が体系に含まれる場合、S_N法による詳細計算中に$\left(\Sigma_{\mathrm{t},g,i,j,k}-\alpha/v_g\right)$の項が負の値をとる可能性がある。この問題により、$S_N$法による transport sweep の結果得られる流出角度中性子束やメッシュ平均の全中性子束も負となり、拡散加速計算の不安定性を招くことことがわかった。そこで、delta-tracking 法に基づき非等方な仮想散乱源を計算に導入し$\left(\Sigma_{\mathrm{t},g,i,j,k}-\alpha/v_g\right)$の項を$\left(\Sigma_{\mathrm{t},g,i,j,k}-\alpha/v_g+\alpha_{\mathrm{est}}/v_g\right)$と補正することで反復中の$S_N$法及び拡散加速計算の数値不安定性の改善を図った。

第 3 章では、GPGPU を用いたS_N法に基づくα固有値計算コードの開発について述べた。

まず、GPU を科学技術計算などに応用する技術である GPGPU の概要を説明した。次に、kernel と呼ばれる全 thread に共通したコードを GPU で実行するという CUDA プログラミングモデルを示した。また、CUDA における grid, block, warp からなる thread の階層構造と、32 thread からなる warp が同一命令を同時に実行する SIMT アーキテクチャについて説明した。加えて、CUDA における主に global, shared memory からなるメモリの階層構造と、メモリアクセスの仕組みとメモリアクセス時に意識すべき点、GPGPU を用いた核計算コード開発において一般に課題となる事柄について述べた。

次に、GPGPU を用いたS_N法に基づくα固有値計算の実装について述べた。まず全体の計算フローを示した後、効率的なメモリアクセスを意識した非等方散乱中性子源計算と transport sweep の実装手法を示した。本研究では、GPU の性能を十分に利用するために、transport sweep を空間メッシュについて並列化する手法である tiled hyperplane transport sweep を用いた。そして、計算に必要だが高コストを要する global memory に対する `atomicAdd()` 関数の呼び出しを減らすため、version 1, 2, 3 とコードを改善する手法を示した。version 3 では、shared memory を利用したうえで、各 thread が計算する展開次数(l,m)の組をずらすことで必要な `atomicAdd()`関数の呼び出し回数を削減した。そして、性能測定の結果 version 1 と比較して約 2.8 倍の高速化と、優れた性能を示した version 3 を採用した。また、これらの計算に必要となるメモリの量を概算し、今回の実験解析の典型的な計算条件下において global memory 消費量は搭載メモリ容量の 1/10 以下、block あたりの shared memory 消費量は上限の約 1/3 であり、メモリ消費量が現実的な範囲内に収まっていることを確認した。

次に、GPGPU を用いてα固有値拡散加速計算を実装する手法について述べた。多くの計算は GPU に対応した線形代数ライブラリである cuBLAS 及び MAGMA を用いて実装でき、中性子源の計算には、S_N法の非等方散乱中性子源計算の手法を流用できることを述べた。また、連立一次方程式の数値解法には、並列化との親和性が高い点ヤコビ前処理を適用した BiCGSTAB 法を採用した。そして、拡散加速計算に必要となるメモリの量を概算、典型的な計算条件下における消費量はS_N法の計算に必要な量と合わせても、搭載メモリ容量の約 1/9 であり、消費量が実行可能な範囲内に収まっていることを確認した。

第 4 章では、第 2 章及び第 3 章に基づき開発した、GPGPU を用いたS_N法に基づくα固有値計算コードの妥当性や高速化の効果を検証した。

まず、妥当性確認及び検証計算では、先行研究においてパルス中性子法によるαが測定された水槽体系を計算の対象とすることと、計算コードに与えた計算条件を示した。そして、先行研究によるα実験値と、開発した CPU 及び GPU コードによるαの計算結果を比較することで、開発したコードの妥当性確認及び検証を実施した。4.3.1 項では、CPU コードによる計算値$\alpha_{\mathrm{calc,CPU}}$と GPU コードによる計算値$\alpha_{\mathrm{calc,GPU}}$が収束条件と同程度の範囲で一致していることから、GPU コードが正しく実装できていることが検証できた。また、過去実験の実験値α_{exp}とα_{calc}がよく一致することから、本研究で作成したコードの妥当性を確認することもできた。

一方で、水槽体系が小さくなるにつれα_{exp}とα_{calc}の差異が大きくなっており、その原因を考察するため、TSL に起因するα計算値の不確かさを決定論的サンプリング法に基づいて評価した。その結果、α_{exp}とα_{calc}の差異は水の ^{1}H TSL データに起因するα不確かさ2σの範囲内に含まれており、水の TSL データがα_{calc}に大きな影響を及ぼしている可能性があることが分かった。

最後に、開発した CPU 及び GPU コードの計算時間を比較し、GPU コードの有効性を検証した。CPU 及び GPU は、それぞれ Intel Core i9-10980XE 及び NVIDIA RTX4090 を使用した。エネルギー172 群、$(ND, NL) = (72, 3)$の条件下での最大寸法水槽体系(空間メッシュ分割数36^3)についての計算で、2160 秒から 197 秒まで約 11 倍の高速化が GPU コードにより達成できたことを確認した。一方で、小さな寸法の水槽体系についての計算では GPU コードによる高速化率が低下しており、これは計算時間の多くを占める拡散加速計算において発行される thread が少なく、GPU の性能を使い切れていないためである可能性があると考察した。また、$(ND, NL) = (1200, 7)$のようなより複雑で大きな問題では、発行 thread 数の増加により、さらなる高速化が期待できることを確認した。

以上で述べた検討結果より本研究では、GPGPU を利用した高速な計算が可能な、非等方散乱中性子源を考慮したS_N法に基づくα固有値計算コードを開発することができたといえる。本研究は、αを利用した核データ調整を始めとして、今後の原子炉物理学及び核計算分野の発展に貢献するものであると考える。

5.2　今後の課題

今後の課題として、以下のことが挙げられる。

- 拡散加速計算のさらなる高速化

4.4.2 項で述べたように、開発した GPU コードでは拡散加速計算が総計算時間の 9 割以上を占めるボトルネックとなっているため、拡散加速計算のさらなる高速化は今後の課題といえる。高速化の方法としては、例えば拡散加速計算にさらに収束加速を施すことが挙げられる。本コードでは拡散加速計算の収束に数十～数百回ほどの外部反復を要しているものの、各回の計算は数ミリ秒～数百ミリ秒と十分に高速である。ゆえに、拡散加速計算に対して空間均質化あるいはエネルギー群縮約を施した CMFD 加速法[1]を適用することで外部反復回数を削減し、より高速な拡散加速計算を実現できると考えられる。

- host・device 間データ転送の最適化

4.4.2 項で述べたように、GPU コードでは host・device 間のデータ転送に transport sweep の処理時間に比べて約 10 倍の時間を要している。これは、拡散加速計算における中性子流補正係数D_{cor}や係数行列$\mathbf{A}$の計算をはじめとして、host(CPU)側により処理を実施する箇所が複数あり、それに伴い host・device 間で全中性子束や中性子流などのデータの転送を行っていることに起因する。計算全体では拡散加速計算がボトルネックであるため、データ転送時間による影響は小さいが、拡散加速計算をさらに高速化し他の処理と処理時間が同程度のオーダーにできた場合には、データ転送はボトルネックとなり得る。したがって、host・device 間のデータ転送の最適化は今後の課題といえる。改善するには、host 側で行っている処理を device 側でできるようにすることでデータ転送を最小限にする必要がある。また、複数のデータをひとまとめにして転送するようにすることでオーバーヘッドを低減することによっても改善できると考えられる。

- 半精度・単精度浮動小数点数の活用

4.2 節で述べたように、本研究では、メモリアクセスが主なボトルネックであったことと計算精度や数値安定性の観点からすべての浮動小数点数演算に倍精度浮動小数点数(FP64)を利用した。しかし、近年の GPU は画像処理や機械学習向けに単精度演算性能(FP32)に特化し、FP64 演算器を FP32 演算器数の$1/64$程度のみ搭載しているものが多い。また、機械学習にさらに特化するために 16 bit で表される半精度浮動小数点数(FP16)が利用できる演算器を搭載した GPU も登場しつつある。ゆえに、GPU の性能をさらに活かすには FP16, FP32 演算の活用が必須と言える。

一方で、科学技術計算では計算精度が求められるため、一般的には FP64 演算が利用される。そこで、近年では FP16, FP32, FP64 演算の混合利用が盛んに試みられている。例えば、中性子輸送計算では不可分操作などの負荷の高い操作を FP32 で行うことで高速化を

試みた例[2]が存在する。また、計算の初期は FP32 を利用し、収束付近では FP64 を用いるという方法も考えられる。加えて、入力を FP16、出力を FP32 とする演算器[3]や、そのような計算において精度を補正技術[4]も開発されており、これにより FP32 程度の精度を保ちつつ高いパフォーマンスを発揮することができるとされている。

これらの技術を用いることで GPU によるさらなる計算の高速化が期待できることから、半精度・単精度浮動小数点数の活用が今後の課題として挙げられる。

5.3 参考文献

[1] N. Z. CHO, C. J. PARK, “A Comparison of Coarse Mesh Rebalance (CMR) and Coarse Mesh Finite Difference (CMFD) Acceleration Methods for the Neutron Transport Calculations,” Proc. M&C 2003, Gatlinburg, Tennessee, April 6–11, 2003, American Nuclear Society (2003);

[2] P. SONG, Z. Zhang, L. LIANG, et al., “Implementation and performance analysis of the massively parallel method of characteristics based on GPU,” Annals of Nuclear Energy 131, 257 (2019); https://doi.org/10.1016/j.anucene.2019.02.026.

[3] “NVIDIA Tensor Cores: Versatility for HPC & AI,” NVIDIA; https://www.nvidia.com/en-us/data-center/tensor-cores/; (current as of Jan. 5, 2024).

[4] H. OOTOMO and R. YOKOTA, “Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance” (2022); https://doi.org/10.48550/ARXIV.2203.03341.

口頭発表

[1] 山口響, 遠藤知弘, 山本章夫, “GPU を活用した拡散理論に基づくα固有値計算,” 日本原子力学会 2023 年春の年会, 1K09, Tokyo, Japan, 3 月 13–15 日 (2023).

[2] 山口響, 遠藤知弘, 山本章夫, “GPU 拡散加速を適用したS_N法によるα固有値計算,” 日本原子力学会 2023 年秋の大会, 1M10, Nagoya, Japan, 9 月 6–8 日 (2023).

[3] H. Yamaguchi, T. Endo, A. Yamamoto, “α-eigenvalue Calculation using the S_N Method with GPU Diffusion Acceleration,” *Proc. RPHA2023*, B-2-9, Gyeongju, Korea, Oct. 24–26 (2023).

[4] 山口響, 遠藤知弘, 山本章夫, “非等方散乱中性子源を考慮したS_N法に基づくα固有値計算の GPU による高速化,” 日本原子力学会 2024 年春の年会, Higashiosaka, Japan, 3 月 26–28 日 (2024). (submitted)