課題研究報告書

前処理付き GMRES 法の GPU に対する適合性調査

伊藤 健一

主指導教員　井口 寧

北陸先端科学技術大学院大学
先端科学技術研究科
（情報科学）

令和 4 年 9 月

概 要

数値シミュレーションでは、物理現象を記述する偏微分方程式を有限差分法や有限要素法などで離散化することで得られる**大規模かつ疎な連立一次方程式** $A\boldsymbol{x} = \boldsymbol{b}$ を求解することになる。クラスタやスーパーコンピュータといった大規模な計算機ではなく、小規模ながら高い演算性能を持つ **GPU** を活用して、オンサイト環境での数値シミュレーションを行う試みが行われている。例えば、医療分野では情報管理やシミュレーションのコントロールの点から外部組織の計算機よりも自組織の計算機での計算が望まれる場合がある。

アクセラレータとして GPU を用いるが、CPU と GPU では特性が異なるため、適したアルゴリズムも異なることが予想される。そのため、本研究では大規模かつ疎な連立一次方程式の GPU に適した数値解法を明らかにすべく、GPU (NVIDIA A100 PCIe) と CPU (HPC System "KAGAYAKI") で前処理付き GMRES 法の評価を行った。

連立一次方程式の求解手法である反復解法は、適当に選んだ初期値から出発して、真の解に収束していく近似解の列を逐次作成していく手法である。反復解法として**クリロフ (Krylov) 部分空間**に基づく**一般化最小残差 (GMRES) 法**がある。GMRES 法は右辺ベクトル $\boldsymbol{b}$ から出発してクリロフ部分空間を拡大しながら正規直交基底 $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_{j+1}$ を生成し、各ステップにおいて残差ノルム $\|\boldsymbol{r}_j\|_2 = \|\boldsymbol{b} - A\boldsymbol{x}_j\|_2$ が最小になるように近似解 $\boldsymbol{x}_j$ を更新していく手法である。

クリロフ部分空間法の収束性は一般に係数行列 A の固有値分布に依存し、固有値分布が少なくかつ 1 (単位行列) に近いほど収束が早い。収束性や安定性の改善を目的として反復に入る前にあらかじめ係数行列 A に対し**オーダリング**、**前処理**を施す。

GPU 上で GMRES 法での反復中に使われる直交化について**古典グラムシュミット 2 (CGS2) 法**と**修正グラムシュミット (MGS) 法**の比較を行った。CUDA による評価の結果、GPU 上での CGS2 法は MGS 法と比べても収束性は問題なく処理時間が大幅に短いということが分かった。

不完全 LU 分解 (ILU) 前処理付き GMRES 法について ILU(0) or ILU(1) or ILU(2) の評価を行った結果、悪条件の問題の場合は手厚い前処理 (ILU(2) など) が必要になってくることがわかった。また、ILU のフィルインを許すレベルについては、非ゼロ要素数の増加量や収束性などに応じて処理時間が最短となる適切なレベルが異なることがわかった。

オーダリングについては**逆 Cuthill-McKee (RCM)** と **Nested Dissection (ND)** について GMRES 法で評価を行った。求解対象の行列によるが概ね、CPU の場合は RCM を適用することで収束性が向上し処理時間が短縮でき、GPU の場合は RCM を適用することで収束性が向上したが処理時間は伸びた。また、GPU

の場合はオーダリングにNDを用いると求解対象の行列によるが概ね、GMRESの処理時間を短縮できることが分かった。

Abstract

Numerical simulation involves solving **large and sparse simultaneous linear equations** obtained by discretizing partial differential equations describing physical phenomena using the finite difference method or the finite element method. Attempts are being made to perform numerical simulations in an on-site environment by utilizing a **GPU**, which are small in scale but have high computing performance, rather than large-scale computers such as clusters or supercomputers. For example, in the medical field, from the viewpoint of information management and simulation control, it may be desirable to use the computer of the own organization rather than the computer of the external organization.

GPUs are used as accelerators, but since the characteristics of CPUs and GPUs are different, it is expected that the suitable algorithm will also be different. Therefore, this research evaluated the preconditioned GMRES method on a GPU (NVIDIA A100 PCIe) and CPUs (HPC System "KAGAYAKI") to reveal GPU-friendly numerical solution methods for large and sparse simultaneous linear equations, obtained by discretizing differential equations with finite difference methods or finite element methods.

The iterative method, which is for solving simultaneous linear equations, starts from an appropriately chosen initial value and successively creates a sequence of approximate solutions that converge to the true solution. As an iterative method, there is a **Generalized Minimal RESidual (GMRES) method** based on the **Krylov subspace**. The GMRES method starts from the right-hand vector $\boldsymbol{b}$ and generates an orthonormal basis $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_{j+1}$, expanding the Krylov subspace. It updates the approximate solution $\boldsymbol{x}_j$ so that the residual norm $\|\boldsymbol{r}_j\|_2 = \|\boldsymbol{b} - A\boldsymbol{x}_j\|_2$ is minimized at each step.

The convergence of the Krylov subspace method generally depends on the eigenvalue distribution of the coefficient matrix, and the smaller the eigenvalue distribution and the closer it is to 1 (identity matrix), the faster the convergence. To improve convergence and stability, the coefficient matrix is **ordered** and **preconditioned** before starting the iteration.

I compared the **Classical Gram-Schmidt 2 (CGS2)** and **Modified Gram-Schmidt (MGS)** methods, for orthogonalization used during iterations in the GMRES method on the GPU. The results of the evaluation by CUDA show that the CGS2 method on the GPUs has much shorter processing time than the MGS method with no convergence issues.

The **Incomplete LU decomposition (ILU)** preconditioned GMRES method was evaluated for ILU(0), ILU(1), or ILU(2), and it was found that heavy preconditioning (such as ILU(2)) is required for ill-conditioned problems. Regarding the level at which fill-in is allowed in ILU, it was found that the appropriate level at which the processing time is minimized depends on the increase in the number of nonzero elements and convergence.

For ordering, **Reverse Cuthill-McKee (RCM)** and **Nested Dissection (ND)** were evaluated by the GMRES method. Although it depends on the matrix to be solved, in general, the application of RCM on CPUs improved convergence and reduced processing time, while the application of RCM on GPUs improved convergence but increased processing time. In the case of GPUs, it was found that using ND for ordering reduced the GMRES processing time in general, although it depends on the matrix to be solved.

目 次

図 目 次

表 目 次

アルゴリズム目次

コード目次

第1章　緒言

1.1　研究の背景

数値シミュレーションでは、物理現象を記述する偏微分方程式を有限差分法や有限要素法などで離散化することで得られる大規模で疎な連立一次方程式を求解することになる。この膨大な計算をクラスタやスーパーコンピュータといった大規模な計算機ではなく、小規模ながら高い演算性能を持つ GPU を活用して、オンサイト環境で行う試みが行われている。**有限差分法 (Finite Difference Method; FDM)** とは、物体における支配方程式 (解析の対象とする現象を記述する方程式で、多くの場合、偏微分方程式) を、領域内に定めた格子点で差分表示することで解く方法である。**有限要素法 (Finite Element Method; FEM)** とは、連続体をいくつかの要素に分けて考え、要素ごとに方程式を作り、それをもとに全体としての方程式を組み立て解く方法である。

GPU (Graphics Processing Units) は元々グラフィックス向けのハードウェアであったが、その高い並列性を生かして画像処理以外の用途でも盛んに使われるようになった。GPU を用いた計算のことを **GPGPU (General-Purpose computing on GPU)**[*1]や **GPU コンピューティング**と呼ぶ。高性能計算 (High-Performance Computing; HPC) 分野の需要に合わせて、グラフィックスや深層学習用途では不要な倍精度浮動小数点演算の性能が強化されたり、データのエラーをチェック・修正できる ECC メモリを搭載したモデル (NVIDIA 社の GPU ではデータセンター向けの Tesla シリーズ) が発売されている。GPU の搭載メモリ量は 2016 年に発売された NVIDIA Tesla P100 では 16GB であったのが、2021 年に発売された NVIDIA A100 では 80GB 搭載するようになっており、大規模な行列が扱いやすくなっている。また、スーパーコンピュータの性能ランキングである TOP500[*2]を見ると、GPU を搭載したシステムが増加してきているため、GPU での処理を高速化することに意義がある。

有限要素法は構造力学や流体力学などの様々な分野で使用され、心臓シミュレーションや機械設計などに用いられている。高精度な数値シミュレーションにはクラスタやスーパーコンピュータなどの大規模な計算機を使用することが多いが、コストが高いため外部組織のものを借りてシミュレーションを行うことが見受けら

[*1] 従来のグラフィックス API とグラフィックス・パイプラインを用いて汎用的な演算処理に GPU を利用することを GPGPU と呼ぶこともある [5, 付録 C]

[*2] https://www.top500.org/

れる。医療分野では情報管理の強化や医師がシミュレーションを管理したいという要請があるため、自組織内での計算資源を利用したシミュレーション実行が望まれる場合がある。そのため、省スペースなワークステーションに搭載可能な GPU などで現実的な時間で解析ができるように高速化することに意義がある。

解くべき連立一次方程式は $A\boldsymbol{x} = \boldsymbol{b}$ で、ただし**係数行列 (coefficient matrix)** $A \in \mathbb{R}^{n \times n}$, A は正則 (regular), **右辺ベクトル**[*3] $\boldsymbol{b} \in \mathbb{R}^n$, **未知ベクトル**[*4] $\boldsymbol{x} \in \mathbb{R}^n$ である。本研究では有限差分法や有限要素法から出てくる、係数行列 A がゼロ要素で大半を占める大規模な**疎行列 (sparse matrix)** である場合を対象とする。

1.2 研究の目的

大規模な連立一次方程式の数値解法として反復法である GMRES 法において、疎行列を対象とした計算の高速化、および数値シミュレーションを行う計算機のダウンサイジング化が大枠としての目的である。アクセラレータとして GPU を用いるが、CPU と GPU では特性が異なるため、適したアルゴリズムも異なることが予想される。本研究では偏微分方程式を有限差分法や有限要素法で離散化して得られる連立一次方程式を倍精度浮動小数点で高速に解ける GPU に適した、GMRES 法で使われているグラムシュミット直交化、および、オーダリング、前処理を明らかにする。そのために、最新の HPC 向けの NVIDIA A100 GPU[*5] と本学の HPC System “KAGAYAKI”[*6] を用いて GPU と CPU で比較実験を行い、各手法の GPU への適合性を評価する。

1.3 本報告書の構成

第 2 章では、連立一次方程式の数値解法一般と GMRES 本体と GPU について議論する。第 3 章では、GMRES 法の直交化で用いる古典グラムシュミット (CGS) 法と修正グラムシュミット (MGS) 法について議論する。第 4 章では、前処理付き GMRES 法について、オーダリングは Cuthill-McKee (CM) と Nested Dissection (ND)、前処理は不完全 LU 分解 (ILU) の議論をする。第 5 章では、第 3 章で議論した直交化について GPU で実験と評価をし、第 4 章で議論した前処理とオーダリングについて CPU と GPU で実験と評価を行う。

図 1.1 に全体の処理の流れと本報告書で議論の対象とする処理 (赤色部分) を示す。疎行列ベクトル積 (SpMV) は河村知記の研究で議論されている [6]。

[*3] **定数ベクトル**とも呼ぶ
[*4] **解ベクトル**とも呼ぶ
[*5] https://www.nvidia.com/ja-jp/data-center/a100/
[*6] https://www.jaist.ac.jp/iscenter/mpc/kagayaki/

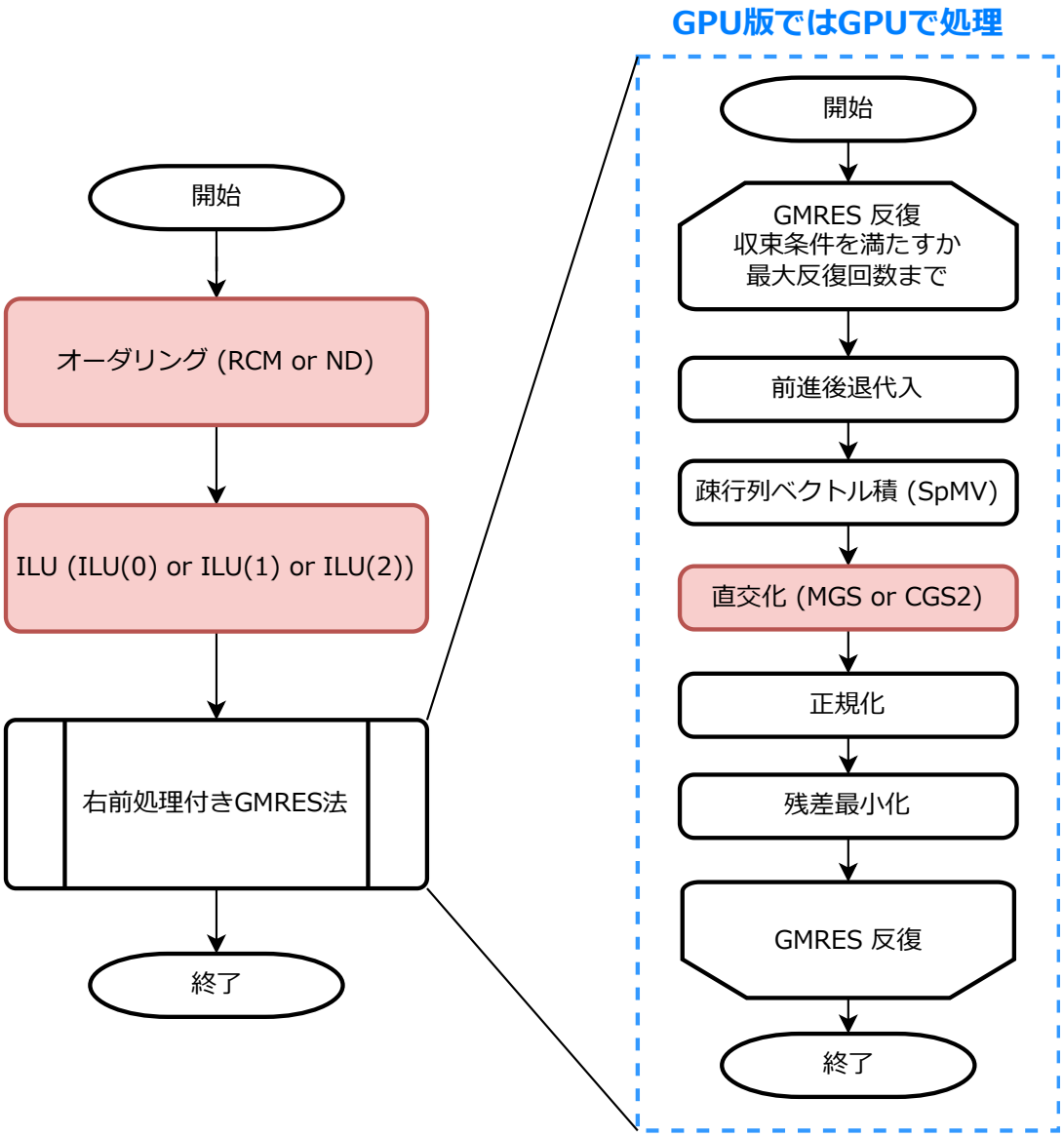

図 1.1: 前処理付き GMRES 法のフローチャート (赤色部分: 本報告書での議論対象の処理)

1.4 表記方法

本報告書では行列、ベクトル、スカラーをそれぞれ $A, \boldsymbol{b}, c$ のように大文字の斜体、小文字の斜体太字、小文字の斜体で表記する。行列やベクトルの要素は a_{ij}, b_i のように対応する小文字の斜体で表記する。

ベクトル $\boldsymbol{v} \in \mathbb{R}^n$ について、**2 ノルム (ユークリッドノルム)** を

$$\|\boldsymbol{v}\|_2 := \sqrt{\sum_{i=1}^{n} |v_i|^2} \tag{1.1}$$

と定義し、**内積 (スカラー積)** を $(\boldsymbol{v}, \boldsymbol{v})$ と表記する。

第2章　連立一次方程式の数値解法

2.1　はじめに

連立一次方程式の代表的な数値解法には大きく分けて**直接解法 (direct method)**[*1]と**反復解法 (iterative method)**[*2]がある。また、係数行列 A の非ゼロ要素数 nnz の割合によって**密行列 (dense matrix)** か**疎行列 (sparse matrix)**、方程式の解きにくさを表す**条件数 (condition number)** によって**良条件 (well-conditioned)** か**悪条件 (ill-conditioned)** という分類があり、最適な解法が異なってくる [1, p.1]。

本章では連立一次方程式の数値解法の俯瞰的な説明と GPU について基本的な説明を述べる。

2.2　直接解法

直接解法 (direct method) は、係数行列 A の変形や代入計算を通じて解の全要素を直接的に求めていく手法である。

行列 A が n 行 n 列である場合、直接解法の計算量は $O(n^3)$ となる。直接解法の利点としては、有限回の演算で解が求まるため所要時間がほぼ予測可能、悪条件の問題に強いという点がある [7, p.6]。

疎行列演算の計算量は、行列の非ゼロ要素数 nnz に比例し、行列の行数 n と列数 n にも比例する。ただし、非ゼロ要素数 nnz とゼロ要素数の和、すなわち全要素数 $n * n$ には依存してない。疎行列演算にかかる処理時間は一般に非ゼロ要素数の演算回数に比例する。[8]

直接解法の代表的な手法にはガウスの消去法や**LU 分解法**がある。解きたい行列が疎行列である場合、LU 分解後の行列は非ゼロ要素が大幅に増加し、メモリ容量が反復解法に比べたくさん必要になる。また、LU 分解の計算には依存性があり、並列化を行うのが難しい。

[*1] **直接法**とも呼ぶ
[*2] **反復法**とも呼ぶ

2.2.1 LU 分解法

LU 分解法 (Lower-Upper decomposition method) は、行列 A を対角要素より上の要素が 0 である下三角行列 L と、対角要素より下の要素が 0 である上三角行列 U に **LU 分解** ($A = LU$) して (式 (2.1), アルゴリズム 2.2.1)、**前進代入 (forward substitution)** $L\boldsymbol{y} = \boldsymbol{b}$、**後退代入 (backward substitution)** $U\boldsymbol{x} = \boldsymbol{y}$ を順に計算する手法である。前進代入では上から順に、後退代入では下から順に方程式を解いていくので、メモリへの書き込みと読み出しが同時に発生する可能性のある**データ依存性 (data dependency)** が生じる。

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ l_{n1} & \cdots & l_{n,n-1} & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & u_{nn} \end{pmatrix} \tag{2.1}$$

アルゴリズム 2.2.1: LU 分解 (IKJ 形式) [1, p.16, Algorithm 2]

```
1: for i = 1, ..., n do
2:     for k = 1, ..., i − 1 do
3:         a_ik := a_ik / a_kk
4:         for j = k + 1, ..., n do
5:             a_ij := a_ij − a_ik * a_kj
6:         end for
7:     end for
8: end for
```

係数行列 A が密行列の場合、LU 分解の計算量は $O(n^3)$ で、前進代入・後退代入の計算量は $O(n^2)$ である。

2.3 反復解法

反復解法 (iterative method) は、適当に選んだ初期解ベクトル $\boldsymbol{x}_0$ から出発して、真の解に収束していく近似解ベクトル $\boldsymbol{x}_j$ の列を逐次作成していく手法である。

収束判定条件 (convergence criterion) には誤差基準と残差基準があり、**誤差 (error) 基準**では修正量が閾値以下になると反復を停止し、**残差 (residual) 基準**では**残差 (residual)** $\boldsymbol{r}_j := \boldsymbol{b} - A\boldsymbol{x}_j$ が閾値以下になるまで反復を繰り返す [9]。

一般的に反復解法にかかる演算時間、メモリ使用量はうまく収束すれば直接解法に比べ少ないと言われている。係数行列 A の変形を行わないため、行列の疎性

すなわちゼロ要素が多いことを利用できる。そのため、係数行列 A が疎である大規模な連立一次方程式の数値計算を行う際に、直接解法よりも反復解法を用いた方がメリットが大きい。一方で、極端に悪条件の問題では疎行列向けの直接解法で連立一次方程式を求解した方が良い場合もある。

反復解法の欠点としては収束性がアプリケーションや境界条件の影響を受けやすく収束しない場合があることが挙げられる。そのため、第 4 章で述べる**前処理 (Preconditioning)** が重要になってくる [10, p.37]。

反復解法は、$\boldsymbol{x}_j \to \boldsymbol{x}_{j+1}$ の漸化式が線形の場合は**定常反復法**、非線形の場合は**非定常反復法**に分類できる。

2.3.1 定常反復法

定常 (stationary) 反復法は線形の漸化式 $\boldsymbol{x}_{j+1} = C\boldsymbol{x}_j + \boldsymbol{d}$ (C: 定数行列, $\boldsymbol{d}$: 定数ベクトル) により近似解を更新していく反復解法である。反復の計算中に解ベクトル $\boldsymbol{x}$ 以外の変数が変化しない。収束率は C の絶対値最大の固有値に依存する。代表的な手法にはヤコビ (Jacobi) 法、ガウス-ザイデル (Gauss-Seidel) 法、SOR(逐次過緩和) 法などがある。

2.3.2 非定常反復法

拘束・最適化条件が加わった**非定常 (nonstationary) 反復法**は、$\boldsymbol{x}_j \to \boldsymbol{x}_{j+1}$ の漸化式が $\boldsymbol{x}_j$ に関して非線形・非定常な反復解法である。一般的に定常反復法よりも収束が速い。

代表的な手法には、**クリロフ部分空間法**に基づく 1952 年に提案された**共役勾配 (Conjugate Gradient; CG) 法**[*3]や 1986 年に提案された **GMRES 法** [11][2, p.171]、チェビシェフ準反復法などがある。

初期近似解ベクトル $\boldsymbol{x}_0$ と対応する**初期残差ベクトル**を $\boldsymbol{r}_0 := \boldsymbol{b} - A\boldsymbol{x}_0$ とするとき、係数行列 A と初期残差ベクトル $\boldsymbol{r}_0$ によって生成される j 次の**クリロフ部分空間 (Krylov subspace)** とは、

$$K_j(A; \boldsymbol{r}_0) := \mathrm{span}\{\boldsymbol{r}_0, A\boldsymbol{r}_0, A^2\boldsymbol{r}_0, \ldots, A^{j-1}\boldsymbol{r}_0\} \tag{2.2}$$

のことを指す。記号 span{} はカッコ内のベクトルの一次結合で張られる空間を表す。

クリロフ部分空間法では、連立一次方程式

$$A\boldsymbol{x} = \boldsymbol{b},\ A \in \mathbb{R}^{n\times n},\ \boldsymbol{b} \in \mathbb{R}^n,\ \boldsymbol{x} \in \mathbb{R}^n \tag{2.3}$$

[*3]スーパーコンピューターの性能測定のためのベンチマークである HPCG (https://www.hpcg-benchmark.org/) で利用されている

の近似解 $\boldsymbol{x}_j$ を条件

$$\boldsymbol{x}_j - \boldsymbol{x}_0 \in K_j(A; \boldsymbol{r}_0) \tag{2.4}$$

を満たすように生成する。しかし、条件式 (2.4) を満たすだけでは近似解 $\boldsymbol{x}_j$ を一意に決められないので、残差ベクトル $\boldsymbol{r}_j := \boldsymbol{b} - A\boldsymbol{x}_j$ に対しても何らかの条件を課す必要がある。[1, p.34, 1.2.2]

2.3.2.1 GMRES 法

アルゴリズム 2.3.1 に示す Yousef Saad と Martin H. Schultz によって開発された**一般化最小残差 (Generalized Minimal RESidual; GMRES) 法** [11][2, p.171] は、**クリロフ部分空間 (Krylov subspace)** の正規直交基底を生成することで近似解 $\boldsymbol{x}_j$ を計算する解法であり、係数行列 A が非対称の場合に使用される。

GMRES 法のアルゴリズムは、初期残差ベクトル $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$ から (初期近似解ベクトル $\boldsymbol{x}_0 = \boldsymbol{0}$ とすると右辺ベクトル $\boldsymbol{b}$ から) 出発して、A をかける操作 (疎行列密ベクトル積) と直交化を繰り返し、クリロフ部分空間を拡大しながら正規直交基底 $\boldsymbol{v}_1, \boldsymbol{v}_2, \ldots, \boldsymbol{v}_{j+1}$ を生成し、各ステップにおいて**残差 (residual)** ノルム $\|\boldsymbol{r}_j\|_2 = \|\boldsymbol{b} - A\boldsymbol{x}_j\|_2$ が最小になるように近似解 $\boldsymbol{x}_j$ を更新していく手法である [12]。**ヘッセンベルグ行列 (Hessenberg matrix)** $\bar{H}_m$ とは、対角要素の 1 つ下まで非ゼロ要素がある行列のことを指す。

アルゴリズム 2.3.1: GMRES [2, p.172, ALGORITHM 6.9]

1: Compute $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$, $\beta := \|\boldsymbol{r}_0\|_2$, and $\boldsymbol{v}_1 := \boldsymbol{r}_0/\beta$
2: **for** $j = 1, 2, \ldots, m$ **do**
3: Compute $\boldsymbol{w}_j := A\boldsymbol{v}_j$ ▷ 疎行列密ベクトル積
4: **for** $i = 1, \ldots, j$ **do** ▷ (修正グラムシュミット) 直交化
5: $h_{ij} := (\boldsymbol{w}_j, \boldsymbol{v}_i)$ ▷ 内積計算
6: $\boldsymbol{w}_j := \boldsymbol{w}_j - h_{ij}\boldsymbol{v}_i$ ▷ 射影を引く
7: **end for**
8: $h_{j+1,j} = \|\boldsymbol{w}_j\|_2$ If $h_{j+1,j} = 0$ set $m := j$ and go to 11 ▷ 2 ノルム計算
9: $\boldsymbol{v}_{j+1} = \boldsymbol{w}_j/h_{j+1,j}$ ▷ 正規化
10: **end for**
11: Define the $(m+1) \times m$ Hessenberg matrix $\bar{H}_m = \{h_{ij}\}_{1 \le i \le m+1, 1 \le j \le m}$
12: Compute $\boldsymbol{y}_m$ the minimizer of $\left\|\beta\boldsymbol{e}_1 - \bar{H}_m\boldsymbol{y}\right\|_2$ and $\boldsymbol{x}_m = \boldsymbol{x}_0 + V_m\boldsymbol{y}_m$

計算の途中で破綻が起きることがなく、残差ノルム $\|\boldsymbol{r}_j\|_2$ が反復回数 j とともに単調に減少するので、非対称行列向けのクリロフ部分空間法の中では、最も頑健 (ロバスト) な方法である。メモリ使用量と 1 反復あたりの演算量が反復回数 j に比例して増加する [7, p.35]。計算量は、密行列の場合 $O(n^2)$ だが疎行列なので非ゼ

ロ要素数 nnz に依存する疎行列密ベクトル積 (SpMV) と、直交化の処理が支配的である。

2.3.2.2 リスタート付き GMRES 法

アルゴリズム 2.3.2 に示す**リスタート付き GMRES 法** [2, p.179] はメモリ資源に制約がある場合に有効な手法である。

リスタート付き GMRES 法は反復を所定のリスタート周期でいったん停止し、得られた近似解ベクトル $\boldsymbol{x}_m$ を初期近似解ベクトル $\boldsymbol{x}_0$ として再びアルゴリズムに適用する手法である。

アルゴリズム 2.3.2: Restarted GMRES [2, p.179, ALGORITHM 6.11]

1: Compute $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0, \beta = \|\boldsymbol{r}_0\|_2$, and $\boldsymbol{v}_1 = \boldsymbol{r}_0/\beta$
2: Generate the Arnoldi basis and the matrix $\bar{H}_m$ using the Arnoldi algorithm starting with $\boldsymbol{v}_1$
3: Compute $\boldsymbol{y}_m$ which minimizes $\left\|\beta\boldsymbol{e}_1 - \bar{H}_m\boldsymbol{y}\right\|_2$ and $\boldsymbol{x}_m = \boldsymbol{x}_0 + V_m\boldsymbol{y}_m$
4: If satisfied then Stop, else set $\boldsymbol{x}_0 := \boldsymbol{x}_m$ and GoTo 1

リスタートすることで保持する正規直交ベクトルの履歴を減らせるため、メモリ使用量は減るが、収束性は悪化する。

2.4 行列の性質

行列の性質は数値解法の収束性に影響するので、本節では文献 [13, p.8〜p.10] を参考に行列の代数的な性質を述べる。

2.4.1 ノルム (norm)

n 次元ベクトル $\boldsymbol{v} \in \mathbb{R}^n$ の p **ノルム** $(1 \leqq p \leqq \infty)$ は

$$\|\boldsymbol{v}\|_p := \begin{cases} \left(\sum_{j=1}^{n} |v_j|^p\right)^{1/p} & (1 \leqq p < \infty) \\ \max_{1 \leqq j \leqq n} |v_j| & (p = \infty) \end{cases} \tag{2.5}$$

で定義される。

任意の行列 $A \in \mathbb{R}^{n \times n}$ に対して、**行列の** p **ノルム**はベクトルのノルムを用いて

$$\|A\|_p := \sup_{\boldsymbol{v} \neq \boldsymbol{0}} \frac{\|A\boldsymbol{v}\|_p}{\|\boldsymbol{v}\|_p} \tag{2.6}$$

と定義される。記号 sup は**上限 (supremum)** である。

行列の 1 ノルムについては

$$\|A\|_1 = \max_{1 \leqq j \leqq n} \sum_{i=1}^{n} |a_{ij}| \tag{2.7}$$

が成り立つ。

2.4.2 特異値、スペクトル半径

行列 $A \in \mathbb{R}^{n \times n}$ に対して、$A^T A$ は対称非負定値行列であり、その n 個の固有値はすべて 0 以上の実数となる。これらの平方根を大きい順に

$$\sigma_1(A) \geqq \sigma_2(A) \geqq \cdots \geqq \sigma_n(A) \geqq 0 \tag{2.8}$$

と書き、A の**特異値 (singular values)** と呼ぶ。[7, p.9]

行列 A の 2 ノルムは A の最大特異値 $\sigma_1(A)$ と等価である。

$$\|A\|_2 = \sigma_1(A) \tag{2.9}$$

行列 A が正則ということは $\sigma_n(A) > 0$ と等価である。

行列の固有値の最大絶対値を、その行列の**スペクトル半径 (spectral radius)** と呼ぶ。

2.4.3 条件数

連立一次方程式をわずかに変えたとき、例えば、右辺ベクトル $\boldsymbol{b}$ に微小な誤差が入った場合など、解が大きく変化することを**悪条件 (ill-conditioned)** という。この悪条件の度合いを測る尺度として条件数がある。[14, p.43][1, p.11〜p.12]

正則行列 $A \in \mathbb{R}^{n \times n}$ に対し、

$$\kappa_p(A) := \|A\|_p \|A^{-1}\|_p \tag{2.10}$$

は p ノルムに関する A の**条件数 (condition number)** と呼ぶ。

2 ノルムに関する A の条件数は

$$\kappa_2(A) = \sigma_1(A)/\sigma_n(A) \tag{2.11}$$

が成り立つ。すなわち、固有値のばらつきが小さいほど、条件数が小さくなる。

疎行列の逆行列は疎行列であるとは限らず、A^{-1} の計算には膨大な演算量が必要になる。条件数の値を正確に求める必要はないため、A^{-1} の値を計算しないで $\|A^{-1}\|$ の値を推定する方法がある。[14, p.45]

単位行列の条件数は 1 である。条件数の大きな連立一次方程式は、残差 $\boldsymbol{r}_m = A\boldsymbol{x}_m - \boldsymbol{b}$ が小さくても近似解 $\boldsymbol{x}_m$ が真の解から遠い場合がある、クリロフ部分空間法が収束しにくいという 2 つの点で解きにくい。[7, p.9][15, p.508]

2.5 疎行列格納形式

有限差分法や有限要素法で得られる係数行列 A はゼロ要素が多い (9 割以上を占める) 疎行列である。そのため、メモリなどへ行列を格納するときに、配列で行列の全要素を記憶するのは計算量とメモリ使用量の面から非効率である。

そこで、非ゼロ要素のみを記憶する**疎行列格納形式 (Sparse Matrix Storage Format)** を用いる。代表的な疎行列格納形式には**座標 (Coordinate; COO) 形式**や**圧縮行格納 (Compressed Row Storage; CRS) 形式**[*4]などがある。疎行列格納形式については文献 [16, 4.3.1 項][6] が詳しい。

疎行列格納形式ごとに対応する演算ルーチンが必要になる。

2.5.1 COO 形式

座標 (Coordinate; COO) 形式とは、非ゼロ要素の (値, 行番号, 列番号) のタプル (組) の集合で行列を表現する形式である。

行列 A を式 (2.12) とすると、

$$A = \begin{pmatrix} a & b & 0 & 0 & 0 \\ c & d & e & 0 & 0 \\ 0 & f & g & h & 0 \\ 0 & 0 & i & j & k \\ 0 & 0 & 0 & l & m \end{pmatrix} \tag{2.12}$$

COO 形式での行列 A は以下のようになり、

```
A.values      = [a, b, c, d, e, f, g, h, i, j, k, l, m] # 値
A.row_indices = [0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 4, 4] # 行番号
A.col_indices = [0, 1, 0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4] # 列番号
```

以下の 3 つの配列で行列を格納する。

- 値を表す (倍精度浮動小数点型) 配列 `values`
- 行番号を表す整数型インデックス配列 `row_indices`
- 列番号を表す整数型インデックス配列 `col_indices`

フォーマットが簡単なため、行列の作成やファイルへの読み書きでよく使われる。その反面、行の境界を判定するのが難しく行単位でスレッド並列化するのが困難なため、計算には向かない。

[*4] 別名 **Compressed Sparse Row (CSR) 形式**

2.5.2 CSR 形式

圧縮行格納 (Compressed Row Storage; CRS) 形式 (**CSR (Compressed Sparse Row) 形式**と呼ぶことが多い) とは行インデックス配列を圧縮する、すなわち行番号を表す配列の代わりに各行の非ゼロ要素の開始位置 (何番目の要素から次の行が始まるか) を表す配列を持つ形式である。

行数 num_rows の行列 A を式 (2.13) とすると、

$$A = \begin{pmatrix} a & b & 0 & 0 & 0 \\ c & d & e & 0 & 0 \\ 0 & f & g & h & 0 \\ 0 & 0 & i & j & k \\ 0 & 0 & 0 & l & m \end{pmatrix} \tag{2.13}$$

CSR 形式での行列 A は以下のようになり、

```
A.values      = [a, b, c, d, e, f, g, h, i, j, k, l, m] # 値
A.row_ptr     = [0, 2, 5, 8, 11, 13]                    # 行先頭ポインタ
A.col_indices = [0, 1, 0, 1, 2, 1, 2, 3, 2, 3, 4, 3, 4] # 列番号
```

以下の 3 つの配列で行列を格納する。

- 値を表す (倍精度浮動小数点型) 配列 values
- 長さ num_rows+1 の、各行の非ゼロ要素の開始位置を示す整数型インデックス配列 row_ptr。ただし、num_rows+1 番目は非ゼロ要素数
- 列番号を表す整数型インデックス配列 col_indices

コード 2.5.1 に CSR 形式での倍精度浮動小数点での **SpMV** (Sparse Matrix – Vector multiplication; 疎行列ベクトル積) の GPU(CUDA) カーネルを示す [3][4]。

コード 2.5.1: 倍精度での CSR(-Scalar) SpMV CUDA kernel [3][4]

```
__global__ void
spmv_csr_scalar_kernel(const int num_rows,
                       const int *row_ptr,
                       const int *col_indices,
                       const double *values,
                       const double *x,
                       double *y)
{
    int row = blockDim.x * blockIdx.x + threadIdx.x;
    if (row < num_rows)
    {
        double dot = 0.0;
        int row_start = row_ptr[row];
        int row_end = row_ptr[row + 1];
        for (int jj = row_start; jj < row_end; jj++)
            dot += values[jj] * x[col_indices[jj]];
        y[row] += dot;
    }
}
```

その他の疎行列格納形式での GPU の SpMV カーネルは文献 [3][4] を参照のこと。

2.6 疎行列ファイル形式

疎行列をファイルに書き出す際のフォーマットとして、Matrix Market Format[*5]、MATLAB .mat ファイル[*6]、Octave's text data format[*7]、Harwell Boeing Format[*8] がある。

Matrix Market Format、Octave's text data format は COO 形式で、Harwell Boeing Format は列方向のインデックスを圧縮する CCS (Compressed Column Storage) 形式で、行列をテキストデータとして格納する。MATLAB .mat ファイルは行列をバイナリデータとして格納する。行列の規模が大きいあるいは非ゼロ要素数が多くなると、テキスト形式だと読み込み時間が計算時間に対して無視で

[*5] https://math.nist.gov/MatrixMarket/
[*6] https://jp.mathworks.com/help/matlab/workspace.html
[*7] https://docs.octave.org/v7.1.0/Simple-File-I_002f0.html
[*8] https://people.sc.fsu.edu/~jburkardt/data/hb/hb.html

きなくなってくるので、バイナリ形式である MATLAB .mat ファイルを利用すると良い。

2.7 CPU と GPU の比較

図 2.1 は CPU と GPU のアーキテクチャの対比である。CPU は**スレッド (thread)** と呼ばれる一連の操作を高速に実行できることに重きを置いて設計されており、数十のスレッドを並列に実行できる。GPU は数千のスレッドを並列に実行できることを重視して設計されており、低いシングルスレッド性能を隠蔽して高い**スループット (throughput)** を達成できる。GPU は高並列な計算に特化しており、データキャッシュやフロー制御ではなくデータ処理に多くのトランジスタを配分するように設計されている。[17]

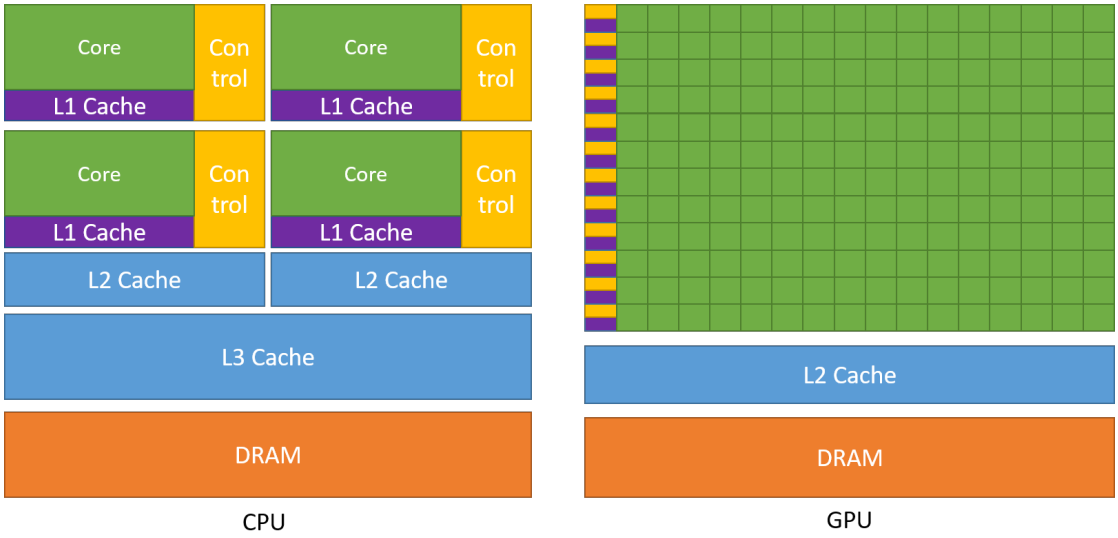

図 2.1: CPU と GPU のチップ資源 (トランジスタ) の配分の違い [17, Figure 1]

コア数を実際に比較してみると、本学の HPC (High Performance Computing; 高性能計算) System “KAGAYAKI”[*9]で用いられている AMD EPYC 7H12 CPU では 1 CPU あたり 64 コア[*10]、1 ノード当たり 2 ソケットで 128 コア存在する。一方で、図 2.2 に示す NVIDIA A100 GPU は 1 GPU あたり CUDA コアが 6,912 個 [18, p.14] 存在する。このように、CPU と GPU でコア数が大幅に異なることが分かる。そのため、GPU と CPU でのプログラムは求められる並列性が異なる。

[*9] https://www.jaist.ac.jp/iscenter/mpc/kagayaki/
[*10] https://www.amd.com/ja/products/cpu/amd-epyc-7h12

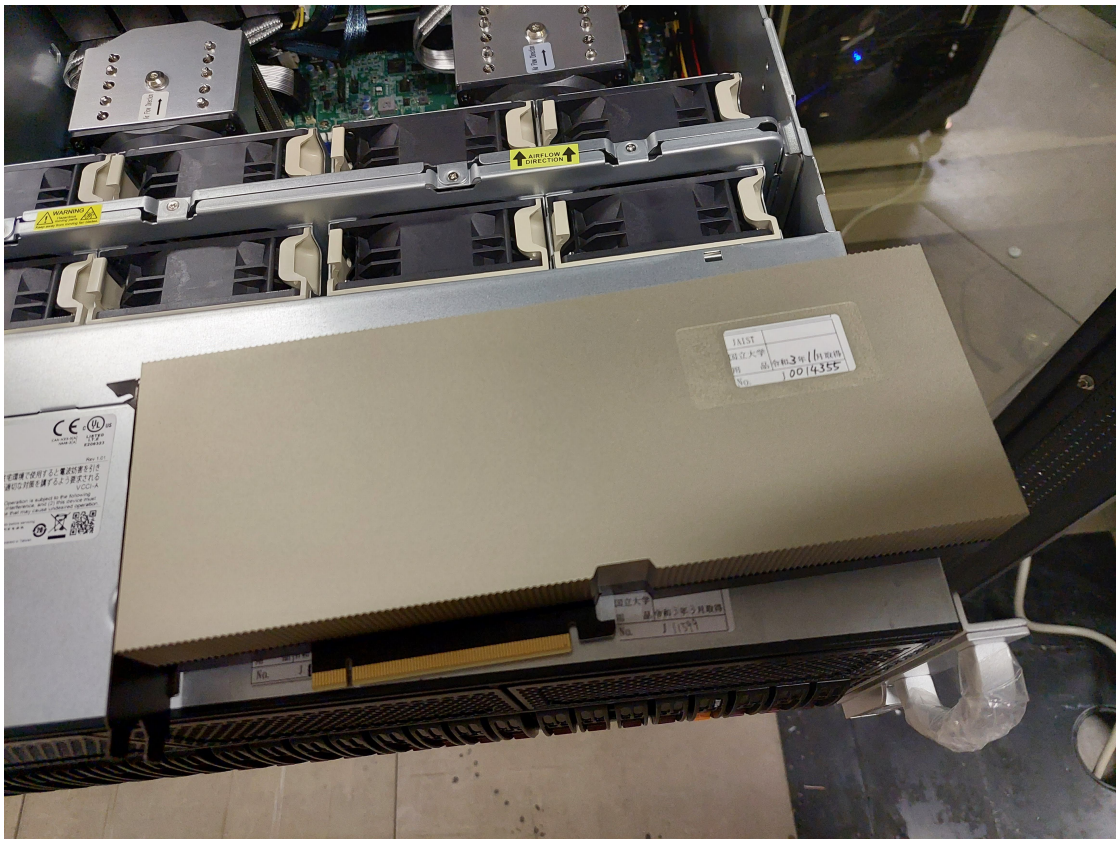

図 2.2: NVIDIA A100 PCIe GPU

図 2.3 に SpMV(疎行列ベクトル積)、BLAS 1 (ベクトル演算)、BLAS 2 (密行列ベクトル演算) などのカーネルのアクセスされたメモリのバイト数に対する浮動小数点演算数の割合である**算術強度 (Arithmetic intensity)** [Flops/Byte] の大まかな範囲を示す。

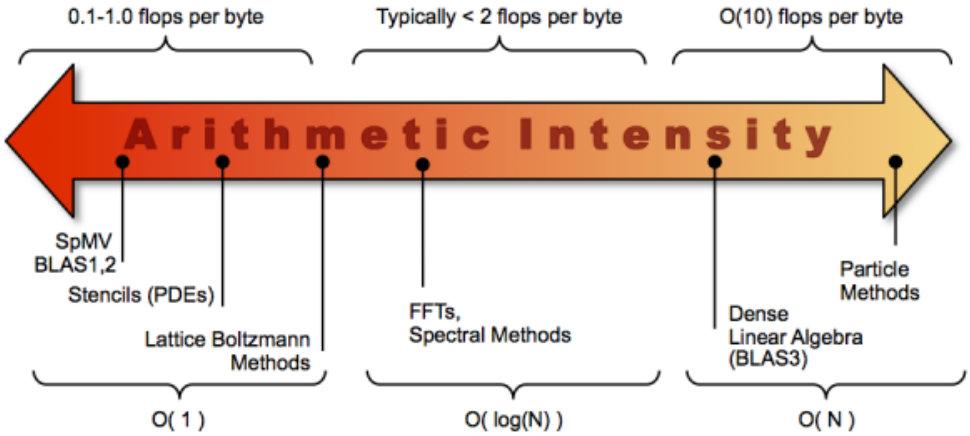

図 2.3: カーネルごとの算術強度 (アクセスされたメモリのバイト数に対する浮動小数点演算数の割合) [19][20]

疎行列を扱うカーネルである SpMV は 2.5.2 項のコード 2.5.1 に示すように間接参照やランダムアクセスが生じ、かつ共役勾配法や GMRES 法といった反復法は SpMV(疎行列ベクトル積)、BLAS 1 (ベクトル演算)、BLAS 2 (密行列ベクトル演算) を多用するので、**算術強度 (Arithmetic intensity)** [Flops/Byte] が低くメモリ帯域幅律速となる。

KAGAYAKI のメモリーは 1 ノード当たり DDR4/3200 SDRAM × 16 チャネルを、NVIDIA A100 40GB のメモリーは HBM2 を、NVIDIA A100 80GB のメモリーは HBM2e を使っている。メモリ帯域幅は理論値で、KAGAYAKI は 1 ノード当たり 409.6 GB/s、A100 40GB PCIe で 1,555 GB/s、A100 80GB PCIe で 1,935 GB/s である。

研究対象のアプリケーションはメモリ帯域幅律速であり、KAGAYAKI の AMD EPYC 7H12 CPU と NVIDIA A100 GPU では A100 GPU の方がメモリ帯域幅が広いので、GPU の方が処理性能が出ることが期待される。

2.7.1 GPU のアーキテクチャ

図 2.4 は Ampere 世代である NVIDIA A100 GPU のアーキテクチャであるが、108 個の Streaming Multiprocessor (SM) を中心に構成されていることがわかる。

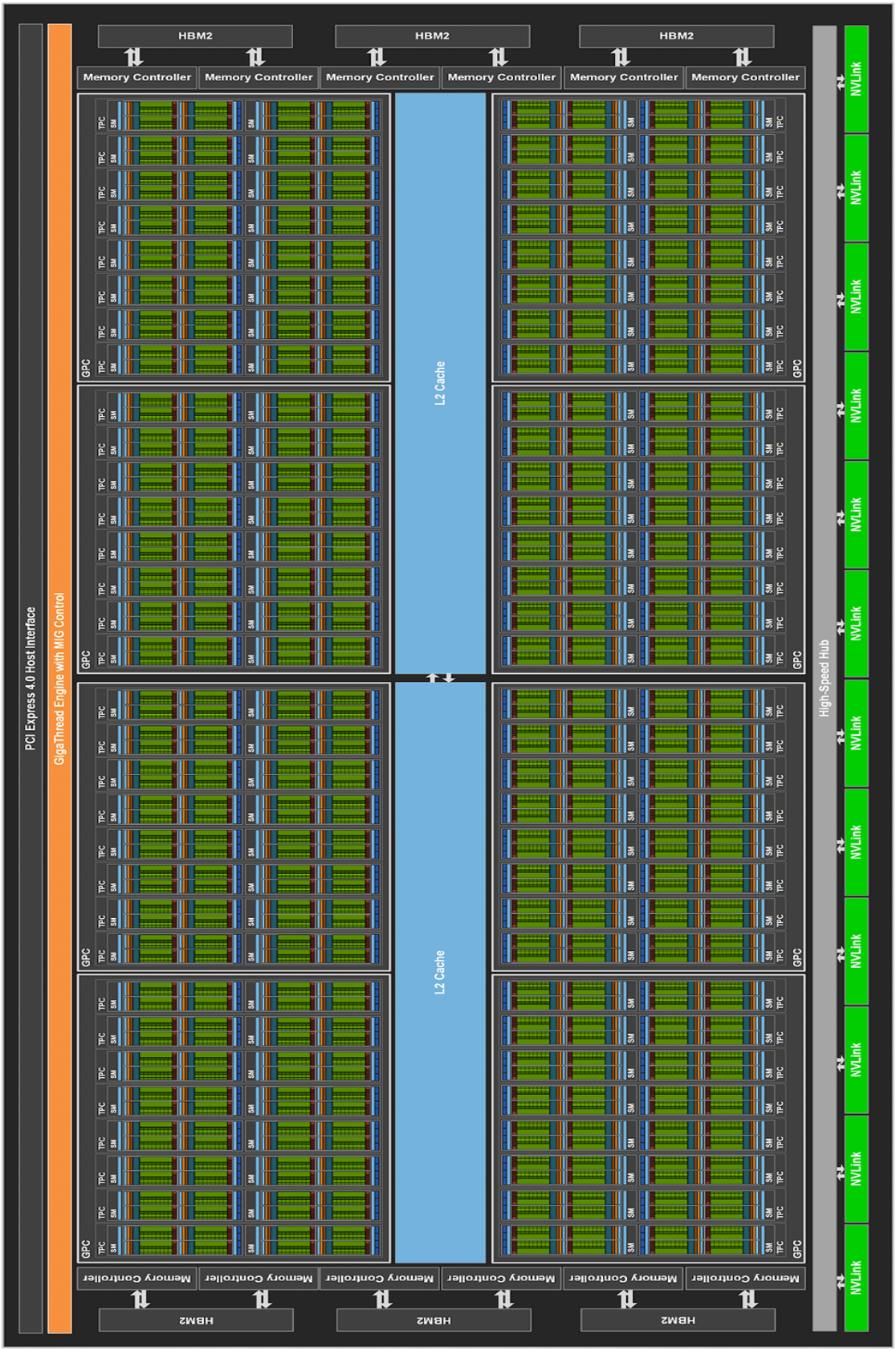

図 2.4: NVIDIA A100 GPU アーキテクチャ [18, p.14, 図 4]

図 2.5 に A100 の Streaming Multiprocessor (SM) のアーキテクチャを示す。

図 2.5: NVIDIA A100 Streaming Multiprocessor (SM) アーキテクチャ [18, p.17, 図 5]

Streaming Multiprocessor (SM) の主な構成要素は以下の通りである。

- CUDA Core (INT32, FP32, FP64)
- Tensor Core
- L1 Cache / Shared Memory
- Load/Store Unit
- SFU (Special Function Unit)
- Warp Scheduler

CUDA Core には INT32、FP32、FP64 の演算器が載っており、Geforce シリーズに比べ Tesla シリーズは FP64 の演算器が充実している。

NVIDIA GPU のアーキテクチャ世代は Fermi → Kepler → Maxwell → Pascal → Volta → Turing → Ampere → Hopper と進んでいっている。

Volta アーキテクチャからの大きな変更点を挙げる。

- CUDA Core の FP ユニットと INT ユニットの分離
- L1 キャッシュと Shared Memory の統合
- 行列積専用演算ユニット Tensor Core の導入

Volta 世代から FP32 ユニットと INT32 ユニットが分離され、浮動小数点 (FP) 演算と整数 (INT) 演算の同時実行が可能になっている [21][22, p.21]。また、Volta 世代から L1 キャッシュと Shared Memory が統合された [22, p.19]。Volta 世代から導入された **Tensor Core** は行列-行列積 (GEMM) の FP16/FP32 混合精度融合積和演算専用ユニットである [22, p.16]。

2.7.2 CUDA について

2006 年に NVIDIA が導入した **CUDA (Compute Unified Device Architecture)** とは、GPU や並列プロセッサ用のスケーラブルな (拡大縮小可能な) 並列プログラミングモデルおよびソフトウェアプラットフォームである。C や C++ を拡張した CUDA C/C++ や Fortran を拡張した CUDA Fortran でプログラムを書くことができる。

2.8 関連研究

石上裕之らは逐次再直交化計算向けの直交化について古典グラムシュミット 2 (CGS2) 法、修正グラムシュミット (MGS) 法およびその他の手法のスレッド並列実装を OpenMP で行い、CPU での評価を行っている [23]。

中島研吾は、ポアソン方程式を有限体積法によって離散化して得られる疎行列を係数行列とする連立一次方程式を、多重格子法による前処理を施した共役勾配法で解いている [24]。オーダリングであるマルチカラー法と逆 Cuthill-McKee (RCM)

法を組み合わた CM-RCM 法を適用して、Flat MPI と OpenMP/MPI ハイブリッド並列を用いた CPU 環境での評価を行っている。

河村知記は疎行列ベクトル積 (SpMV) の GPU での高速化に向けて、行列内のパターン性に着目して圧縮率を向上させた省メモリな疎行列格納形式を提案し、SpMV 単体と GMRES 法での演算時間の評価を行っている [6]。

NVIDIA は前処理に不完全 LU 分解やコレスキー分解を用いた共役勾配 (CG) 法や BiCGStab 法といった反復法の GPU 実装について述べ、CPU と GPU の比較および処理時間の分析を行っている [25]。

Hartwig Anzt らは**しきい値 (threshold)** を用いた並列な不完全 LU 分解 (ParILUT) を提案し [26]、GPU 向けの実装と評価を行っている [27]。ParILUT アルゴリズムは既存の非ゼロ要素のパターンに対する不完全な分解を近似する並列な**不動点反復法 (fixed-point iteration)** と非ゼロ要素のパターンを問題の特性に動的に適応させる戦略を交互に行っている。

Azzam Haidar らは Tensor Core を用いた混合精度での、求められた近似解と真の解との誤差を用いた連立一次方程式を再度解いて解を補正する手法である**反復改良 (Iterative Refinement; IR) 法**の高速化を行っている [28]。

2.9 おわりに

本章では連立一次方程式の数値解法と GPU について議論した結果、研究対象のアプリケーションである疎行列向け反復法 (GMRES 法) はメモリ帯域幅律速であり、kagayaki の CPU と A100 GPU では A100 GPU の方がメモリ帯域幅が広いので、GPU の方が処理性能が出ることが期待できる。

第3章　直交化 (Orthogonalization)

3.1　はじめに

2.3.2.1 目のアルゴリズム 2.3.1 で示した GMRES 法の 4〜7 行目では、クリロフ部分空間の正規直交基底の生成を行う**アーノルディ (Arnoldi) 過程**において逐次再直交化計算をグラムシュミット法で行っている。**直交化 (orthogonalization)** とは複数の一次独立なベクトルの組を正規直交基底に変換するものである。図 3.1 に GMRES 法の処理の流れにおける直交化の位置づけを示す。本章ではグラムシュミット法について議論する。

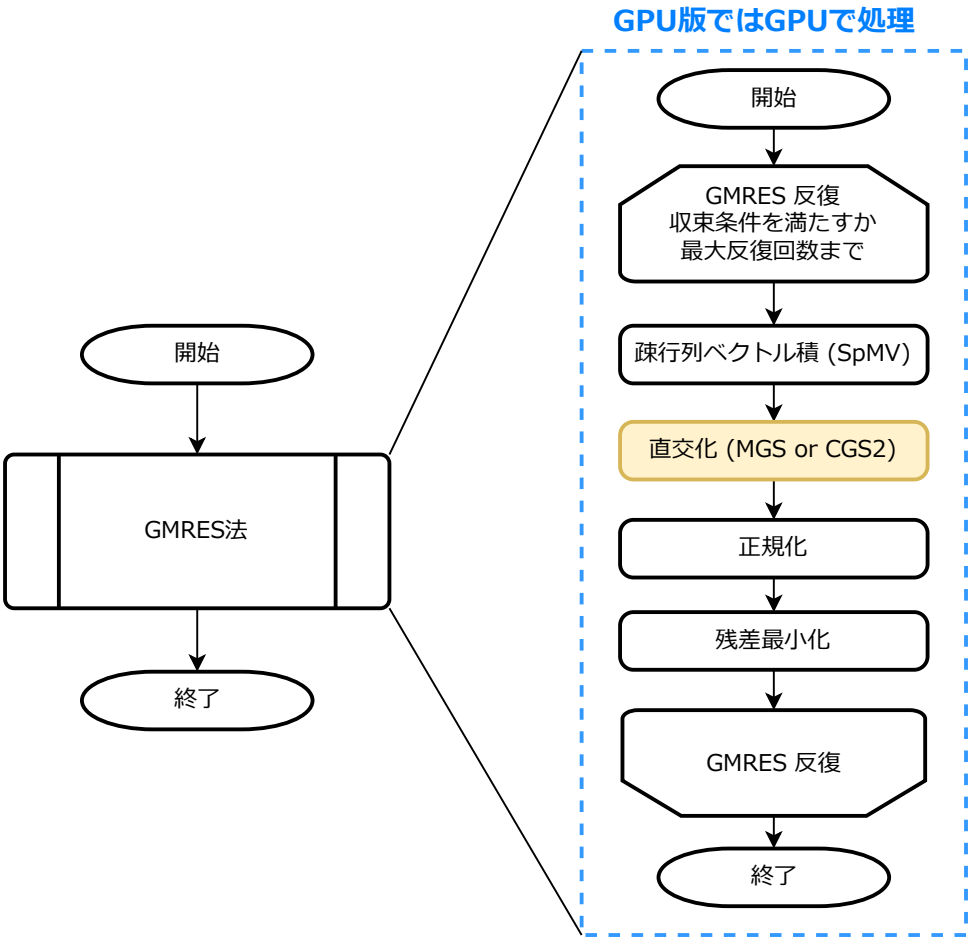

図 3.1: GMRES 法のフローチャートにおける直交化の位置づけ

3.2 グラムシュミット法

逐次再直交化向けの**グラムシュミット (Gram-Schmidt) 法**はベクトルを 1 本ずつ追加していき、それぞれを直交化していく方法である。グラムシュミット法には古典的な計算順序に従う古典グラムシュミット (CGS) 法と、計算誤差の蓄積を考慮した計算順序に従う修正グラムシュミット (MGS) 法がある。

3.2.1 古典グラムシュミット法

アルゴリズム 3.2.1 に示す**古典グラムシュミット (Classical Gram-Schmidt; CGS) 法** [2, p.11] では、アルゴリズム 3.2.1 の 3 行目において新しいベクトル $\boldsymbol{x}_j$ と正規直交基底 $\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_{j-1}$ に対して内積を計算する部分を密行列ベクトル積で行っている。

アルゴリズム 3.2.1: Classical Gram-Schmidt [2, p.11, ALGORITHM 1.1]

1: Compute $r_{11} := \|\boldsymbol{x}_1\|_2$. If $r_{11} = 0$ Stop, else compute $\boldsymbol{q}_1 := \boldsymbol{x}_1 / r_{11}$
2: **for** $j = 2, \ldots, r$ **do**
3: Compute $r_{ij} := (\boldsymbol{x}_j, \boldsymbol{q}_i)$, for $i = 1, 2, \ldots, j-1$. ▷ 密行列ベクトル積
4: $\hat{\boldsymbol{q}} := \boldsymbol{x}_j - \sum_{i=1}^{j-1} r_{ij} \boldsymbol{q}_i$ ▷ 射影を引く
5: $r_{jj} := \|\hat{\boldsymbol{q}}\|_2$, ▷ 2 ノルム計算
6: If $r_{jj} = 0$ then Stop, else $\boldsymbol{q}_j := \hat{\boldsymbol{q}} / r_{jj}$ ▷ 正規化
7: **end for**

CGS 法では密行列ベクトル積 (BLAS 2) で計算可能であるため並列性が良い。しかし、計算精度が MGS 法に比べ劣るため、CGS 法を 2 回実行する **CGS2 法** [29] が提案されている。CGS2 法では CGS 法を 2 回実行するため計算量が CGS 法に比べて 2 倍となる。

3.2.2 修正グラムシュミット法

アルゴリズム 3.2.2 に示す**修正グラムシュミット (Modified Gram-Schmidt; MGS) 法** [2, p.12] ではアルゴリズム 3.2.2 の 4 行目〜7 行目において、新しいベクトル $\boldsymbol{x}_j$ と正規直交基底 $\boldsymbol{q}_1, \boldsymbol{q}_2, \ldots, \boldsymbol{q}_{j-1}$ に対して内積を計算する部分を、計算誤差の蓄積を考慮して逐次的に行っている。

アルゴリズム 3.2.2: Modified Gram-Schmidt [2, p.12, ALGORITHM 1.2]

1: Define $r_{11} := \|\boldsymbol{x}_1\|_2$. If $r_{11} = 0$ Stop, else $\boldsymbol{q}_1 := \boldsymbol{x}_1/r_{11}$.
2: **for** $j = 2, \ldots, r$ **do**
3: Define $\hat{\boldsymbol{q}} := \boldsymbol{x}_j$
4: **for** $i = 1, \ldots, j-1$ **do**
5: $r_{ij} := (\hat{\boldsymbol{q}}, \boldsymbol{q}_i)$ ▷ ベクトル演算 (内積)
6: $\hat{\boldsymbol{q}} := \hat{\boldsymbol{q}} - r_{ij}\boldsymbol{q}_i$ ▷ 射影を引く
7: **end for**
8: Compute $r_{jj} := \|\hat{\boldsymbol{q}}\|_2$, ▷ 2 ノルム計算
9: If $r_{jj} = 0$ then Stop, else $\boldsymbol{q}_j := \hat{\boldsymbol{q}}/r_{jj}$ ▷ 正規化
10: **end for**

MGS 法では計算順序の依存性によりベクトル演算 (BLAS 1) を用いる必要があり、並列性が CGS 法より劣る。石上裕之らは逐次再直交化計算向けの直交化について CGS2 法、 MGS 法およびその他の手法を、CPU で直交化単体の評価をしているが、MGS 法よりも CGS2 法の方が実行時間が短くなっており [23]、MGS 法は GPU では特に処理性能が出にくいと考えられる。

3.3 Basic Linear Algebra Subprograms (BLAS)

BLAS とは密行列を対象とする数値線形代数の演算に必要な関数を定義する API である。

Level 1 ベクトル演算

Level 2 行列ベクトル演算

Level 3 行列同士の演算

高度な最適化が施された実装が多数存在する。代表的な実装として、

- netlib[*1]による公式リファレンス実装: reference BLAS[*2]
- Intel 社による実装: Intel Math Kernel Library (Intel MKL)[*3]
- オープンソース実装: OpenBLAS[*4]
- NVIDIA 社による GPU 向け実装: cuBLAS[*5]

などがある。また、疎行列向けの BLAS である Sparse BLAS[*6]も存在する。

[*1] http://www.netlib.org/
[*2] http://www.netlib.org/blas/
[*3] https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html
[*4] http://www.openblas.net/
[*5] https://developer.nvidia.com/cublas
[*6] https://math.nist.gov/spblas/

古典グラムシュミット法で計算する必要がある (倍精度の) 密行列ベクトル積では一般行列とベクトルの積を計算する dgemv() 関数、修正グラムシュミット法で計算する必要がある (倍精度の) 内積では ddot() 関数、(倍精度の) 2 ノルム計算では dnrm2() 関数を用いている。

3.4 おわりに

本章では GMRES 法の直交化で使われる古典グラムシュミット (CGS) 法と修正グラムシュミット (MGS) 法について議論した。その結果、計算精度が良い修正グラムシュミット (MGS) 法と、古典グラムシュミット (CGS) 法を 2 回実行するため計算量は 2 倍に増えるが並列性は良い古典グラムシュミット 2 (CGS2) 法ではどちらが処理時間が短いか評価するべきという結論に達した。

第4章　前処理 (Preconditioning)

4.1　はじめに

定常反復法の収束性は反復行列の**スペクトル半径 (spectral radius)** に、クリロフ部分空間法の収束性は一般に係数行列 A の固有値分布に依存し [1, p.51]、固有値分布が少なくかつ 1 (単位行列) に近いほど収束が早い。そのため、反復法では収束性や安定性の改善を目的として反復に入る前にあらかじめ係数行列 A に対し**オーダリング (Ordering)**、**前処理 (Preconditioning)** を施すことが多い。

各反復で高速に解け、事前準備のコストが小さく、収束性を大きく改善可能で、高い並列性を持つのが理想的な前処理である。

本章では、4.2 節でオーダリングについて Cuthill-McKee (CM) と Nested Dissection (ND)、4.3 節で前処理について不完全 LU 分解 (ILU) の議論を行う。

4.2　オーダリング (Ordering)

オーダリング (Ordering) とは、適当な置換行列 U を用いて係数行列 A の行と列の同時置換 $A' = UAU^\top$ を行い、元の連立一次方程式 $A\boldsymbol{x} = \boldsymbol{b}$ を $A'\boldsymbol{x}' = \boldsymbol{b}'$ (ただし $\boldsymbol{x}' = U\boldsymbol{x}$, $\boldsymbol{b}' = U\boldsymbol{b}$) に変換することである。

図 4.1 に全体の処理の流れにおけるオーダリングの位置づけを示す。

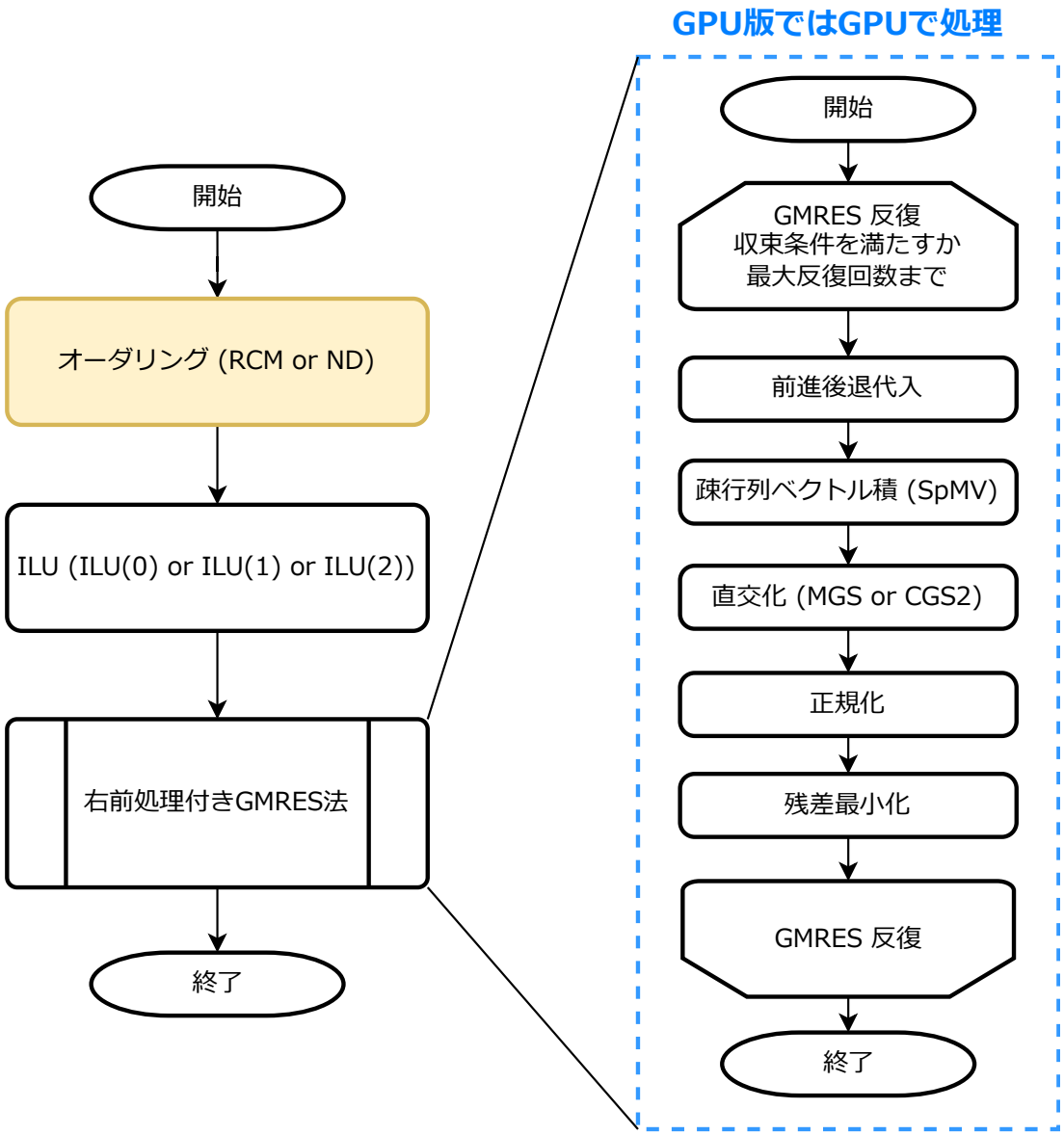

図 4.1: 前処理付き GMRES 法のフローチャートにおけるオーダリングの位置づけ

オーダリングの効果としては以下が挙げられる。[30, p.12]

- データ依存性を排除し、並列性を得る
- ゼロであった要素に非ゼロの値が入る**フィルイン (Fill-in)** を抑制する
- 帯幅やプロファイルを減らす
- ブロック化

ここで用語の定義をしておくと、i 行における非ゼロ成分の列番号の最大値を k とするとき $\beta_i = k - i$ で、**帯幅 (bandwidth)** は β_i の最大値、**プロファイル (profile)** は β_i の和である。帯幅、プロファイル、フィルインともに少ない方が都合が良く、特に帯幅、プロファイルは (前処理付き反復法の) 収束に影響する。[30, p.14]

要素間の接続に関する情報を使って、お互いに依存性を持たない要素群を同じ色に色付け (coloring) することで、色内での並列処理ができるようになる。色付けした後に、色番号の順番に各メッシュを再番号付け (reordering) する [1, p.208]。

4.2.1 逆 Cuthill-McKee (RCM)

本項ではオーダリングの手法である Cuthill-McKee (CM) と逆 Cuthill-McKee (RCM) について述べる。

4.2.1.1 Cuthill-McKee (CM)

アルゴリズム 4.2.1 にレベル集合による並べ替え手法である **Cuthill-McKee (CM)** [31][2, p.84] を示す。CM 法では「レベル (level)」と呼ばれる概念を導入し、レベル内の各要素が依存関係を持たないような条件を付加することで並列計算向きのオーダリングとして利用できる [1, p.210]。CM 法によるオーダリングをすることで行列の帯幅やプロファイルを減少させ、LU 分解でのフィルインを抑制することができる。

アルゴリズム 4.2.1: Cuthill-McKee (G) [2, p.84, ALGORITHM 3.2]

Find an intial node v for the traversal
Initialize $S = \{v\}$, seen $= 1$, $\pi(\text{seen}) = v$; Mark v;
while seen $< n$ **do**
    $S_{\text{new}} = \emptyset$;
    for each node v **do**
        $\pi(++\text{seen}) = v$;
        for each unmarked w in adj(v), going from lowest to highest degree **do**
            Add w to S_{new};
            Mark w;
        end for
        $S := S_{\text{new}}$
    end for
end while

4.2.1.2 逆 Cuthill-McKee (RCM)

逆 Cuthill-McKee (Reverse Cuthill – McKee; RCM) は CM 法を適用した後に、レベル、要素の番号を逆順に振り直す方法である [1, p.210]。RCM 法はフィルインが CM 法よりも少なくなる。

RCM 法を適用した疎行列の非ゼロ要素パターンの可視化を MATLAB を用いて行った。図 4.2 に SuiteSparse Matrix Collection[*1] [32][33] の行列名 F1[*2] の係数行列 A を、図 4.3 に RCM を適用後の行列名 F1 の係数行列 A を示す。RCM によるオーダリングをすることですべての非ゼロ要素が対角要素の近くに移動し、行列の帯幅やプロファイルが小さくなっていることがわかる。

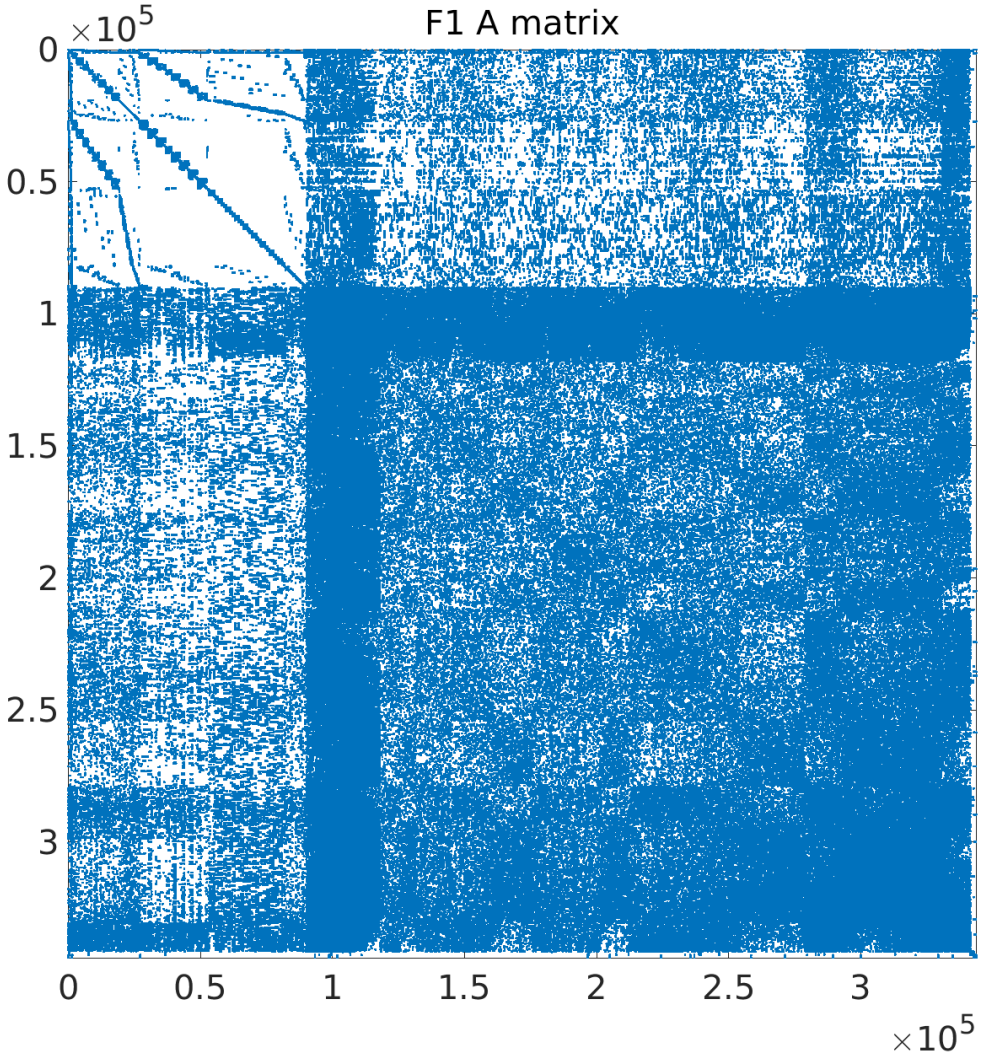

図 4.2: 行列名 F1 の係数行列 A の非ゼロ要素パターン (青色: 非ゼロ要素)
Nonzeros = 26,837,113 (0.023 %)

[*1] https://sparse.tamu.edu/
[*2] https://sparse.tamu.edu/Koutsovasilis/F1

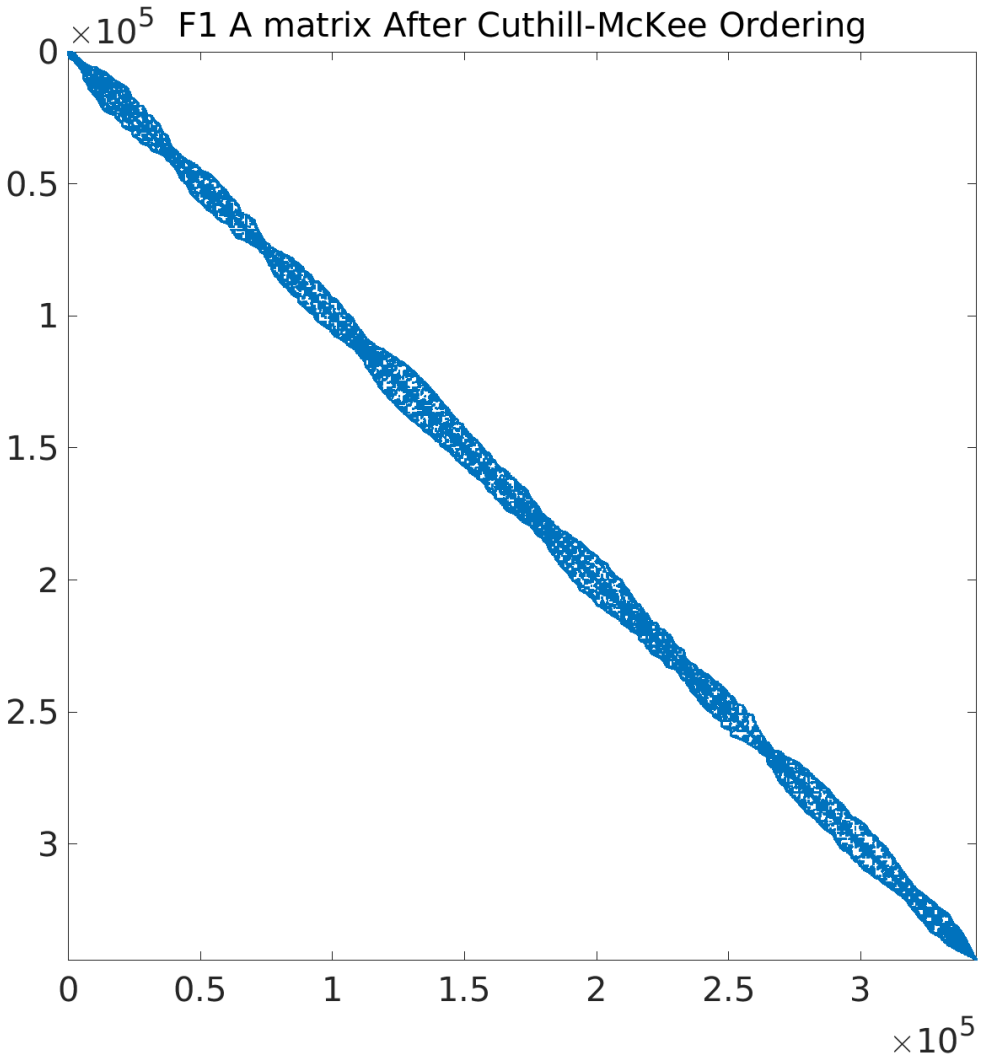

図 4.3: 行列名 F1 の RCM 適用後の係数行列 A の非ゼロ要素パターン
(青色: 非ゼロ要素) Nonzeros = 26,837,113 (0.023 %)

4.2.2 Nested Dissection (ND)

アルゴリズム 4.2.2 に示す **Nested Dissection (ND)** [34][2, p.97, 3.6.2] によるオーダリングは、行列をグラフの隣接行列として扱い、グラフ理論の手法を使用して行列の行と列を並べ替える手法である。グラフ内の頂点と辺を縮約することで問題の規模を大幅に縮小できる。ND 法は主に LU 分解での非ゼロ要素の増加量 (フィルイン) の抑制を目的として用いられる。

アルゴリズム 4.2.2: ND$(G, nmin)$ [2, p.97, ALGORITHM 3.7]

1: **if** $|V| \leq nmin$ **then**
2: label nodes of V
3: **else**
4: Find a separator S for V
5: Label the nodes of S
6: Split V into G_L, G_R by removing S
7: ND$(G_L, nmin)$
8: ND$(G_R, nmin)$
9: **end if**

この手法は再帰性と「セパレーター」の概念を用いることで簡単に説明できる。グラフ内の頂点集合 S は、S を削除するとグラフが 2 つの互いに素な部分グラフに分割される場合に**セパレーター (separator)** と呼ばれる。

アルゴリズム 4.2.2 の 2 行目と 5 行目の頂点のラベル付けは、通常順番に進められる。例えば 5 行目では、S の頂点は手順の中で最後にラベル付けされた頂点から特定の順序でラベル付けされる。ND の手順の主なステップは、グラフを 3 つの部分グラフに分けることであり、そのうちの 2 つの部分グラフは互いに結合していない。3 番目の集合は、最初の集合の両方からの頂点との結合を持ち、セパレーターと呼ばれる。このようにグラフを分割して、各部分グラフで再帰的に処理を繰り返すことが重要な考え方である。セパレーターの頂点には最後の番号が付けられる。[2, p.97, 3.6.2]

図 4.4 は Nested Dissection (ND) による再オーダリング (reordering) をした行列の模式図で、再帰的縁付きブロック対角行列になる。ND 法によるオーダリングで得られる再帰的縁付きブロック対角行列では、各対角ブロックを独立に分解できるため、並列向きのオーダリングとして利用できる [1, p.22]。

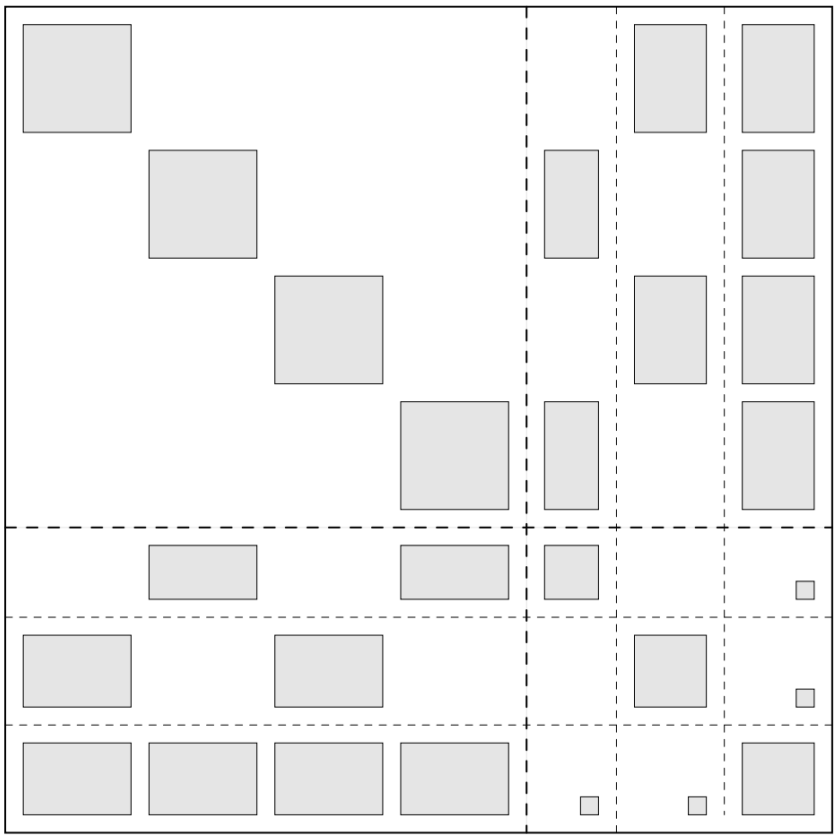

図 4.4: Nested Dissection による再並び替え (reordering) 後の行列の模式図 [2, p.98, Figure 3.11]

Nested Dissection を適用した疎行列の非ゼロ要素パターンの可視化を MATLAB を用いて行った。図 4.5 に Nested Dissection によるオーダリングを適用した後の行列名 F1[*3]の係数行列 A を示す。

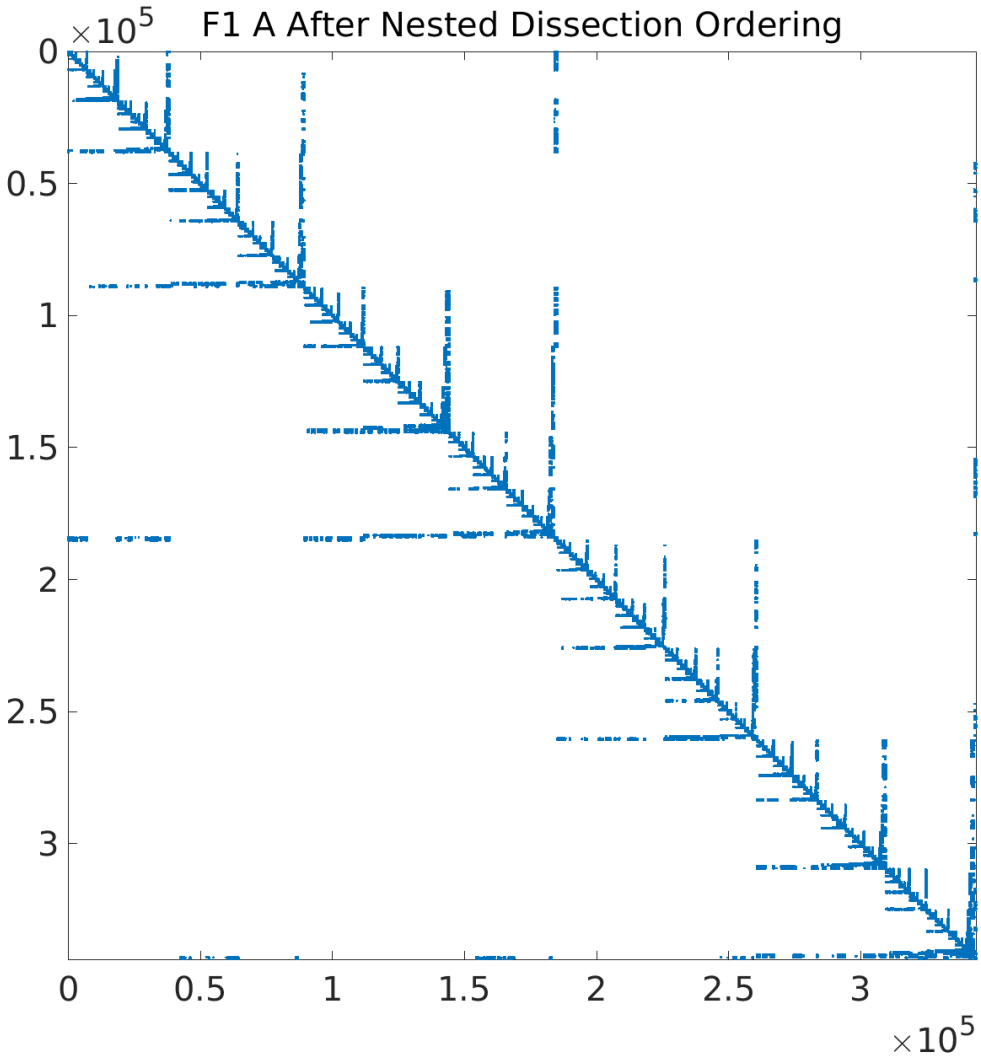

図 4.5: 行列名 F1 の ND 適用後の係数行列 A の非ゼロ要素パターン
(青色: 非ゼロ要素) Nonzeros = 26,837,113 (0.023 %)

[*3] https://sparse.tamu.edu/Koutsovasilis/F1

4.3 不完全 LU 分解 (ILU)

係数行列 A が疎行列の場合、LU 分解 $A = LU$ をすると、元の行列では 0 であった要素に 0 でない値が入る**フィルイン (Fill-in)** が生じ、得られた下三角行列 L と上三角行列 U は非ゼロ要素数が劇的に増加する。

アルゴリズム 4.3.1 に示す**不完全 LU 分解 (Incomplete LU factorization; ILU)** は、LU 分解で生じるフィルインを制限し、近似的な (不完全な) LU 分解 $A \approx \tilde{L}\tilde{U}$ を計算することで、**前処理 (Preconditioning)** に使う手法である。ILU を前処理に使う場合、前処理行列は $M = \tilde{L}\tilde{U}$ となる。

アルゴリズム 4.3.1: ILU 分解 (IKJ 形式) [2, p.305, ALGORITHM 10.3][1, p.57, Algorithm 9]

---

for $i = 2, \ldots, n$ **do** ▷ P: 前処理行列の非ゼロ要素のインデックス集合
  for $k = 1, \ldots, i-1$ and if $(i,k) \notin P$ **do**
    $a_{ik} := a_{ik}/a_{kk}$
    for $j = k+1, \ldots, n$ and for $(i,j) \notin P$ **do**
      $a_{ij} := a_{ij} - a_{ik}a_{kj}$.
    end for
  end for
end for

図 4.6 に全体の処理の流れにおける ILU の位置づけを示す。

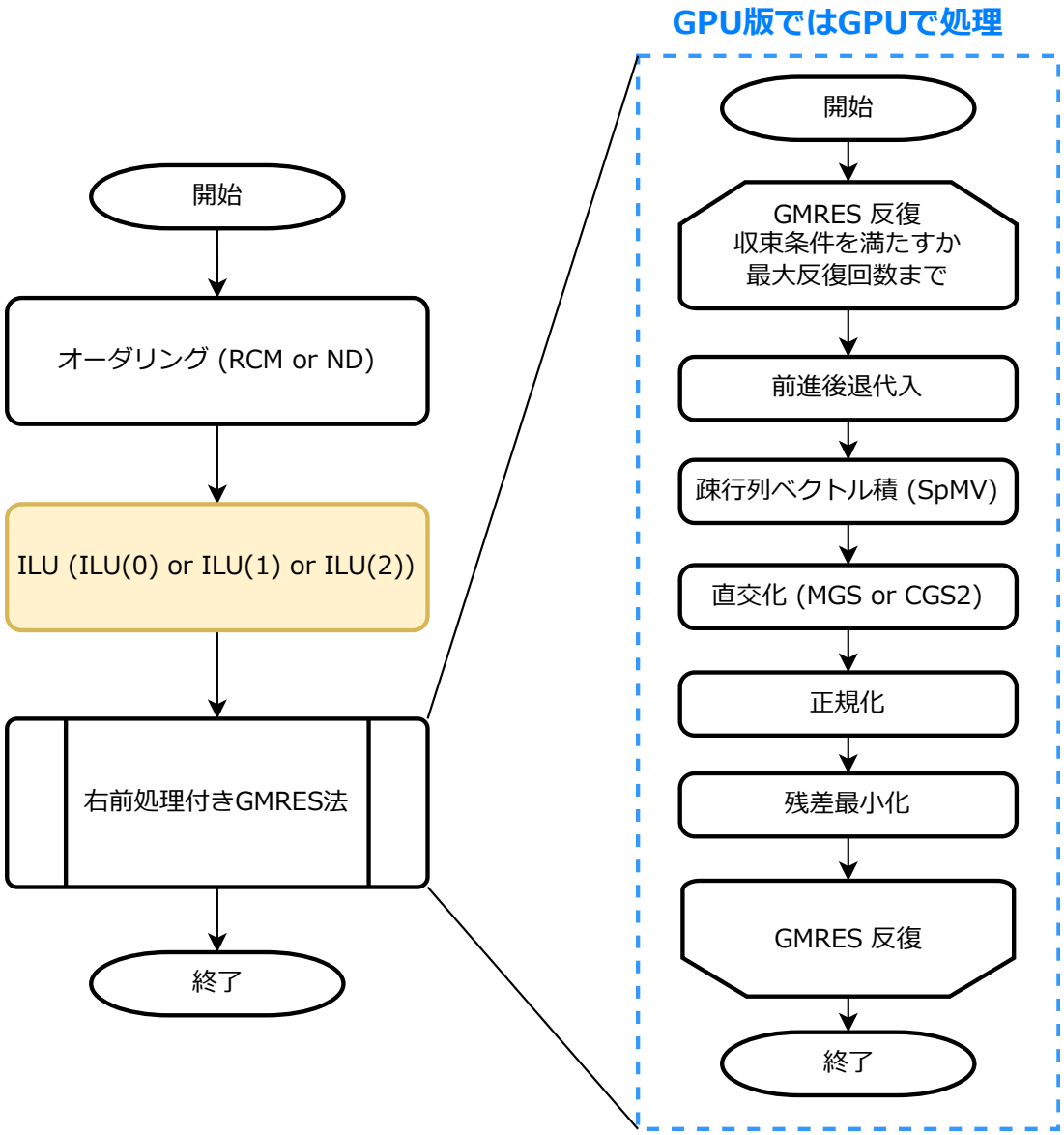

図 4.6: 前処理付き GMRES 法のフローチャートにおける ILU の位置づけ

前処理行列 M の非ゼロ要素のインデックス集合を P とする。前処理行列 M が係数行列 A と等しい構造 $P = \{(i,j) \mid a_{ij} \neq 0\}$ を持つ、すなわち、フィルインを許さないときは **ILU(0)** [2, p.307] と呼ぶ。1 段階のみのフィルインを許すときは **ILU(1)**、2 段階のフィルインを許すときは **ILU(2)** [2, p.311] と表記する。

フィルインを許すレベルが高いほどより正確な前処理行列 M になっていると考えられるため、フィルインを許すレベルが高いほどクリロフ部分空間法の収束性の向上が期待できる。一方で、フィルインを許しすぎると非ゼロ要素数が増えすぎ、演算量やメモリ使用量が増え処理時間が伸びることが懸念される。

4.3.1 右前処理付き GMRES 法

前処理付きの GMRES を実施するにあたり、アルゴリズム 4.3.2 に示す**右前処理付き GMRES (Right Preconditioned GMRES) 法** [2, p.284] を採用した。連立一次方程式 $A\boldsymbol{x} = \boldsymbol{b}$ を

$$AM^{-1}\boldsymbol{y} = \boldsymbol{b},\ \boldsymbol{x} = M^{-1}\boldsymbol{y} \tag{4.1}$$

に変形することを**右前処理**と呼ぶ。この係数行列 AM^{-1} は A に比べて単位行列に近くなっていると考えられるため、クリロフ部分空間法の収束性の向上が期待できる。[1, p.51][7, p.48]

アルゴリズム 4.3.2: GMRES with Right Preconditioning [2, p.284, ALGORITHM 9.5]

1: Compute $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0, \beta = \|\boldsymbol{r}_0\|_2$, and $\boldsymbol{v}_1 = \boldsymbol{r}_0/\beta$
2: **for** $j = 1, \ldots, m$ **do**
3: Compute $\boldsymbol{w} := AM^{-1}\boldsymbol{v}_j$ ▷ M: 前処理行列
4: **for** $i = 1, \ldots, j$ **do** ▷ (修正グラムシュミット) 直交化
5: $h_{i,j} := (\boldsymbol{w}, \boldsymbol{v}_i)$ ▷ 内積計算
6: $\boldsymbol{w} := \boldsymbol{w} - h_{i,j}\boldsymbol{v}_i$ ▷ 射影を引く
7: **end for**
8: Compute $h_{j+1,j} = \|\boldsymbol{w}\|_2$ and $\boldsymbol{v}_{j+1} = \boldsymbol{w}/h_{j+1,j}$ ▷ 正規化
9: Define $V_m := [\boldsymbol{v}_1, \ldots, \boldsymbol{v}_m], \bar{H}_m = \{h_{i,j}\}_{1 \le i \le j+1; 1 \le j \le m}$
10: **end for**
11: Compute $\boldsymbol{y}_m = \mathrm{argmin}_{\boldsymbol{y}} \left\|\beta\boldsymbol{e}_1 - \bar{H}_m\boldsymbol{y}\right\|_2$, and $\boldsymbol{x}_m = \boldsymbol{x}_0 + M^{-1}V_m\boldsymbol{y}_m$
12: If satisfied Stop, else set $\boldsymbol{x}_0 := \boldsymbol{x}_m$ and GoTo 1.

前処理に ILU を用いる場合は反復前に係数行列 A を ILU 分解し、反復中であるアルゴリズム 4.3.2 の 3 行目中の $M^{-1}\boldsymbol{v}_j$ を、係数行列 A を ILU 分解した後の前処理行列 $M = \tilde{L}\tilde{U}$ と $\boldsymbol{v}_j$ について前進・後退代入を行うことで求める。

前処理に ILU を適用することによって反復回数は削減されるが、前処理行列の生成と各反復での前進・後退代入という余分な計算をすることになる。

4.3.2 疎行列向け前進・後退代入

図 4.7 に全体の処理の流れにおける前進後退代入の位置づけを示す。

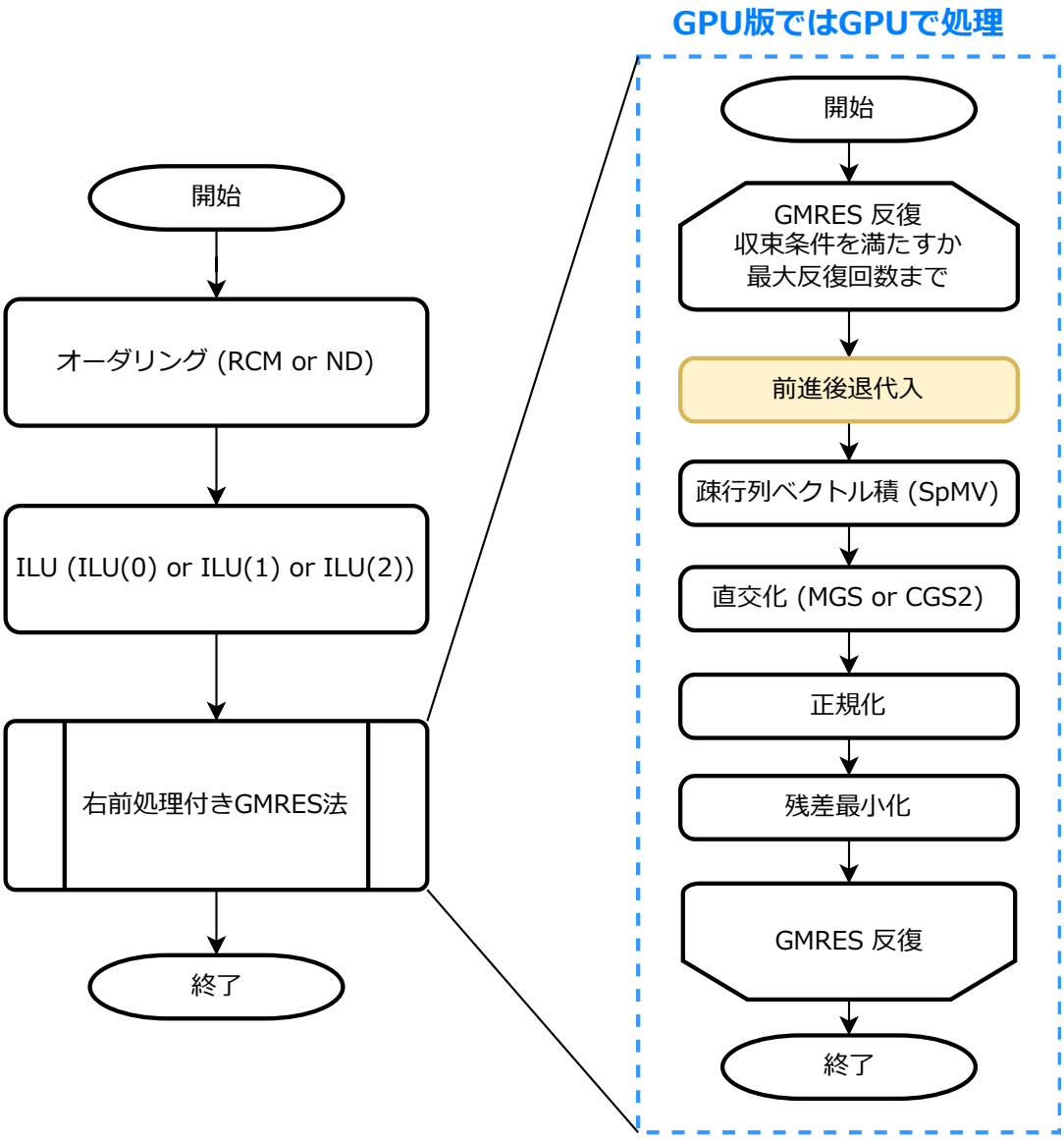

図 4.7: 前処理付き GMRES 法のフローチャートにおける前進後退代入の位置づけ

GPU 上での疎行列向け前進・後退代入の NVIDIA による実装は文献 [35] が詳しい。NVIDIA の実装 [35] では Analysis Phase でデータ依存性を解析した後、Solve Phase で三角行列を解いている。図 4.8 に下三角行列のデータ依存の有向非巡回グラフ (Directed Acyclic Graph; DAG) を示す。一般の行列ではデータ依存の有向非巡回グラフ (DAG) を構築する必要があるが、三角行列では暗黙的なので実際には構築する必要がない。そのため、Analysis Phase では幅優先探索 (Breadth-First-Search; BFS) でデータ依存を走査 (traverse) している。Solve Phase では Analysis

Phase の情報を使って同じレベルに属するノードを並列に処理している。

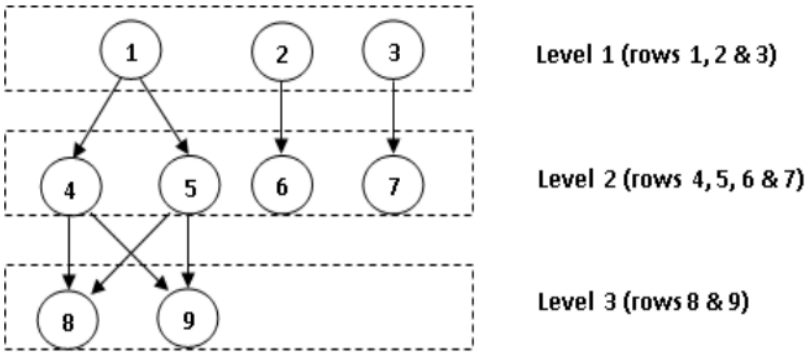

図 4.8: 下三角行列のデータ依存の有向非巡回グラフ (DAG) [35]

4.4 疎行列の非ゼロ要素パターン可視化

図 4.9〜4.17 に 4.2 節で述べたオーダリングについて逆 Cuthill-McKee (RCM) と Nested Dissection (ND)、4.3 節で述べた不完全 LU 分解 (ILU) について ILU(0), ILU(1), ILU(2) の組み合わせで、SuiteSparse Matrix Collection[*4] [32][33] の行列名 consph[*5]に適用した場合の非ゼロ要素パターンを MATLAB で可視化した結果を示す。

RCM によるオーダリングをした場合は行列の帯幅が小さくなっていることが、ND によるオーダリングをした場合は再帰的縁付きブロック対角行列になっていることが確認できる。また、前処理が手厚くなる、すなわち ILU(0) → ILU(1) → ILU(2) になるにしたがってフィルインのレベルが増え、非ゼロ要素数が増加していることが確認できる。

[*4] https://sparse.tamu.edu/
[*5] https://sparse.tamu.edu/Williams/consph

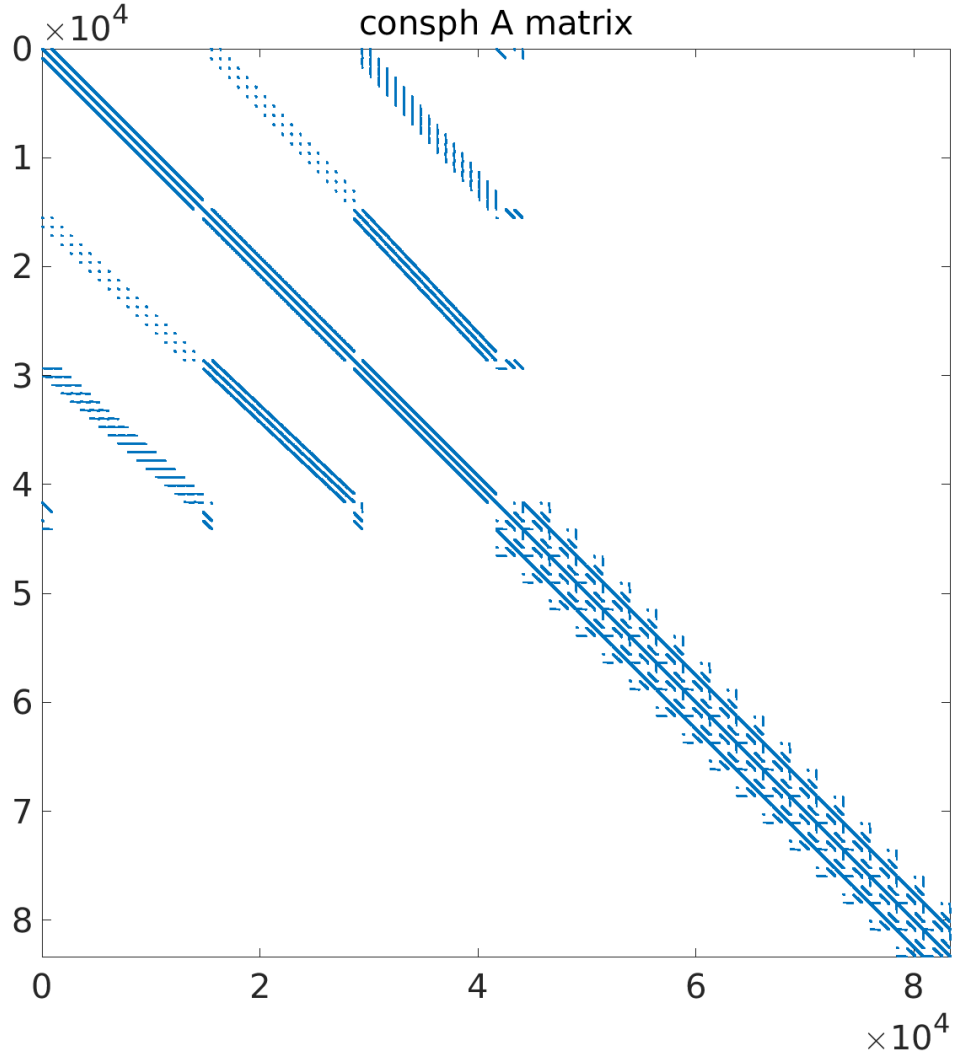

図 4.9: 行列名 consph の係数行列 A の非ゼロパターン (青色: 非ゼロ要素)
Nonzeros = 6,010,480 (0.087 %)

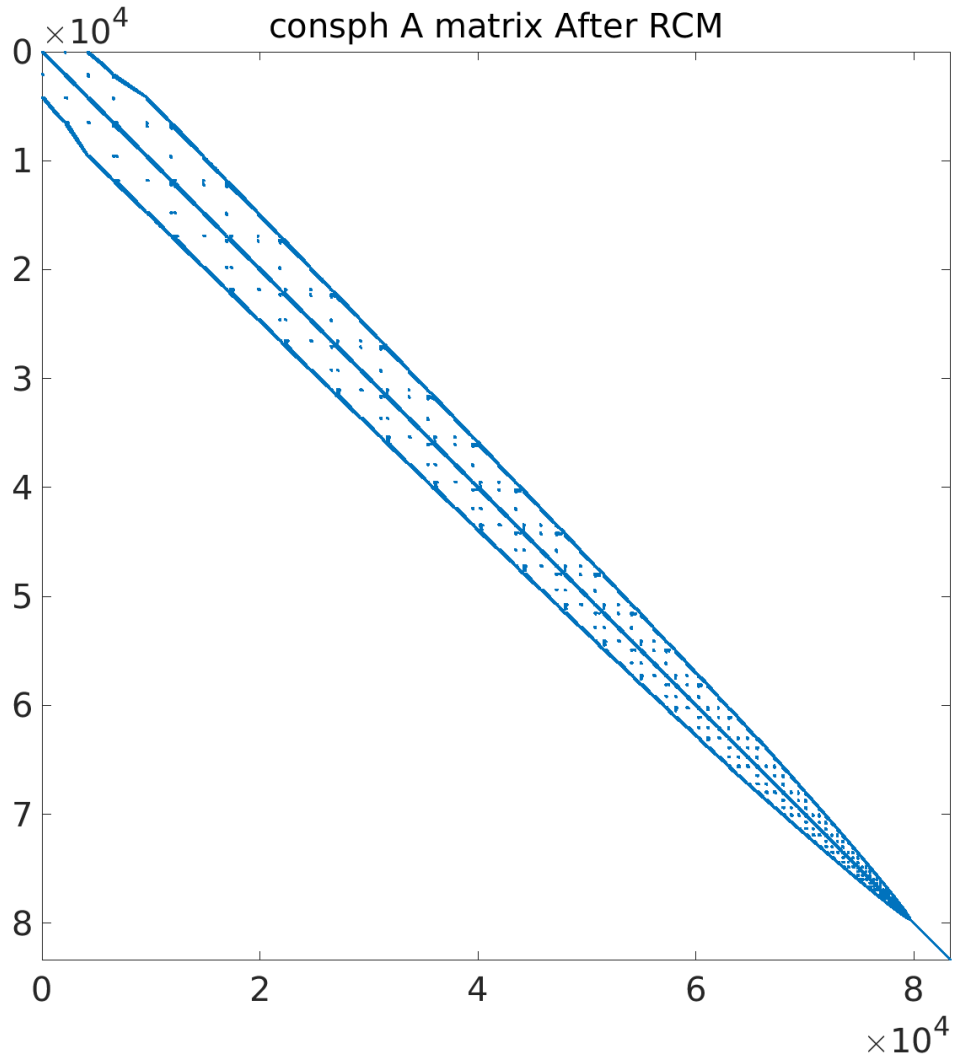

図 4.10: 行列名 consph の RCM 適用後の係数行列 A の非ゼロパターン
(青色: 非ゼロ要素) Nonzeros = 6,010,480 (0.087 %)

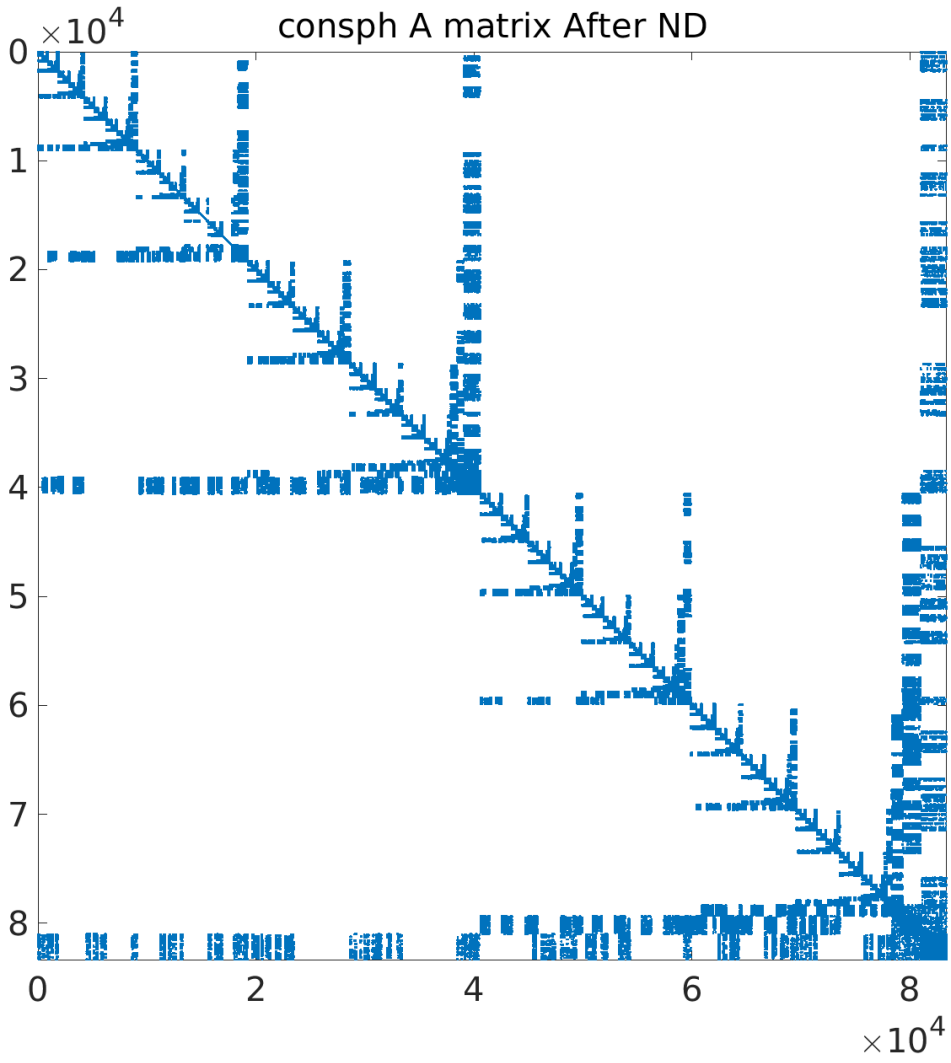

図 4.11: 行列名 consph の ND 適用後の係数行列 A の非ゼロパターン
(青色: 非ゼロ要素) Nonzeros = 6,010,480 (0.087 %)

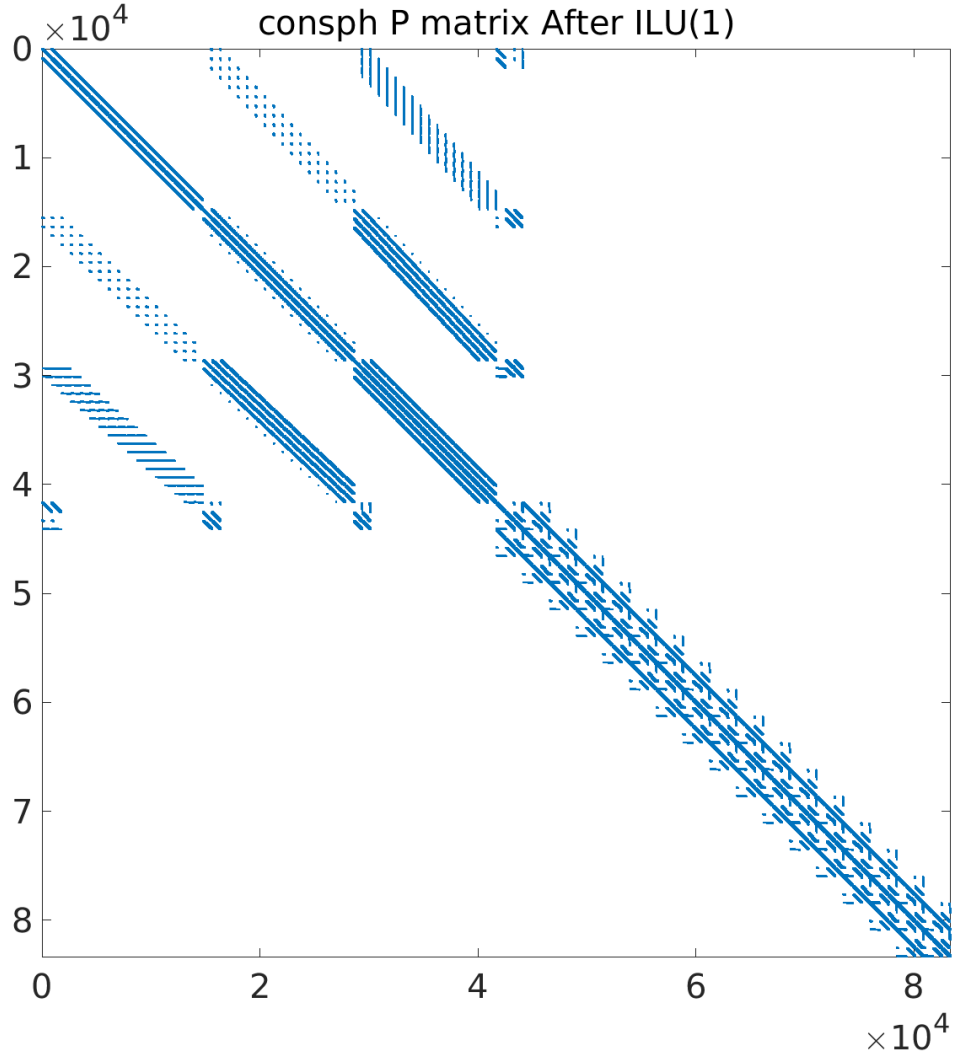

図 4.12: 行列名 consph の ILU(1) 適用後の非ゼロパターン
(青色: 非ゼロ要素) Nonzeros = 13,966,120 (0.201 %)

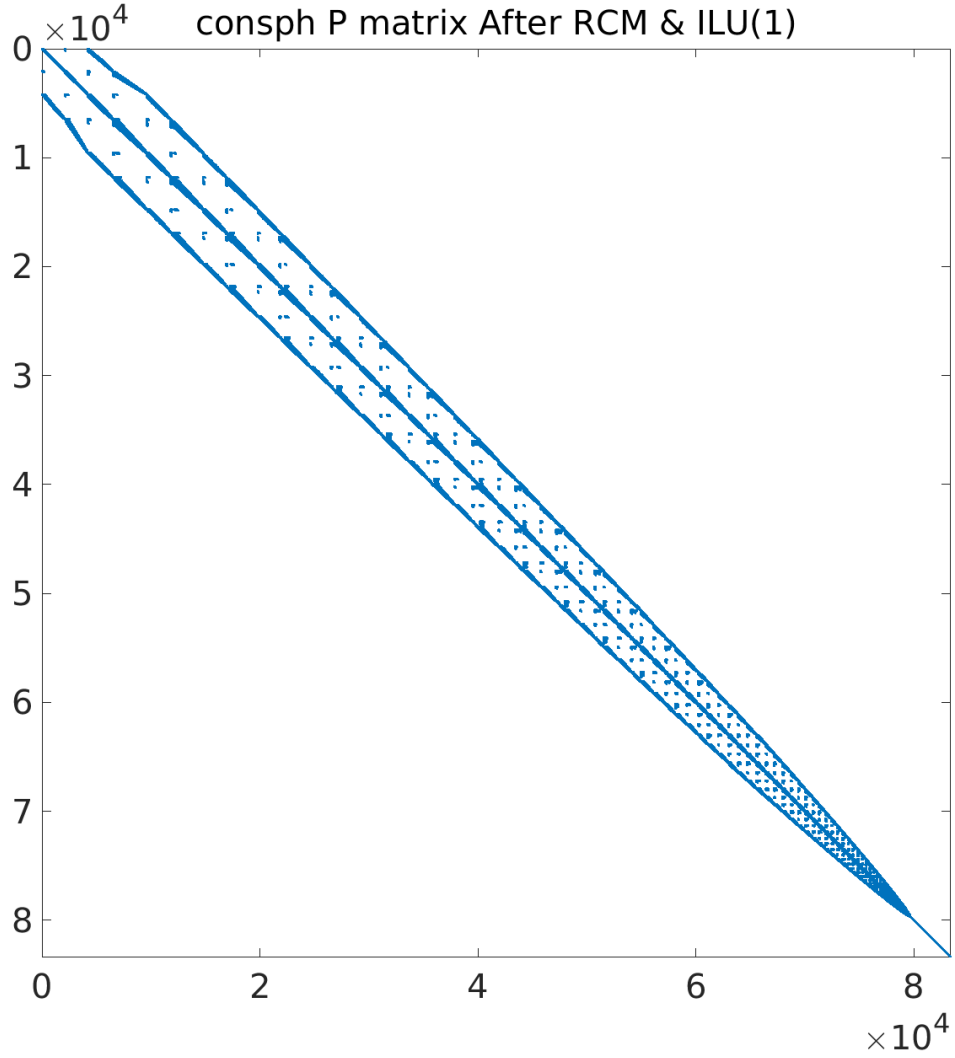

図 4.13: 行列名 consph の RCM と ILU(1) 適用後の非ゼロパターン
(青色: 非ゼロ要素) Nonzeros = 13,443,806 (0.194 %)

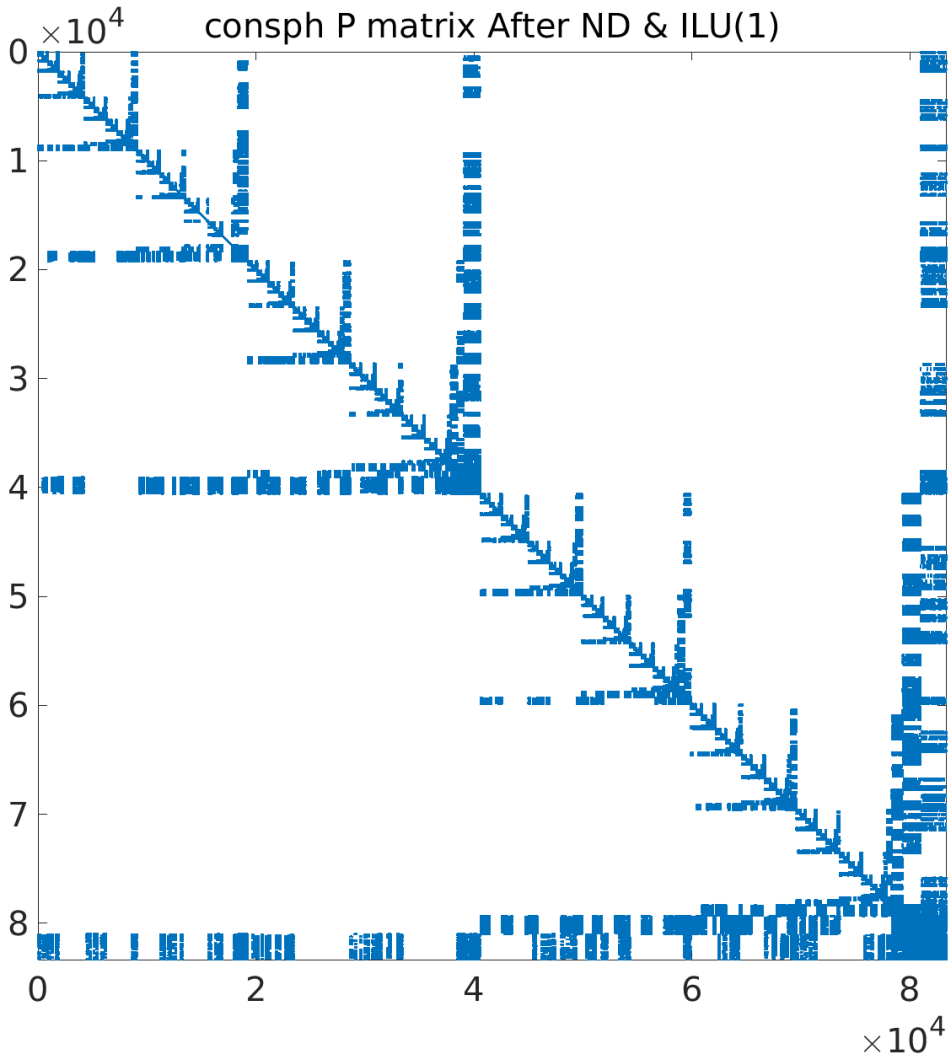

図 4.14: 行列名 consph の ND と ILU(1) 適用後の非ゼロパターン
(青色: 非ゼロ要素) Nonzeros = 16,887,238 (0.243 %)

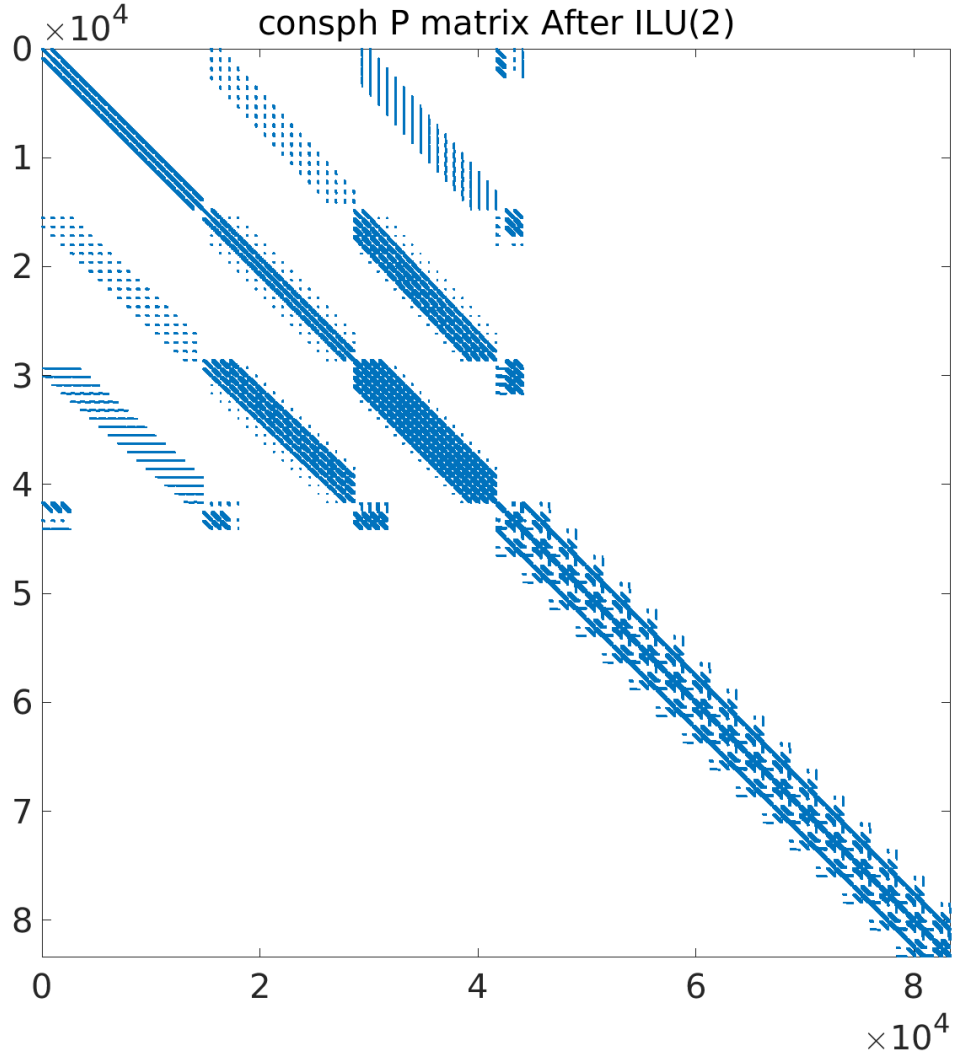

図 4.15: 行列名 consph の ILU(2) 適用後の非ゼロパターン
(青色: 非ゼロ要素) Nonzeros = 34,692,390 (0.500 %)

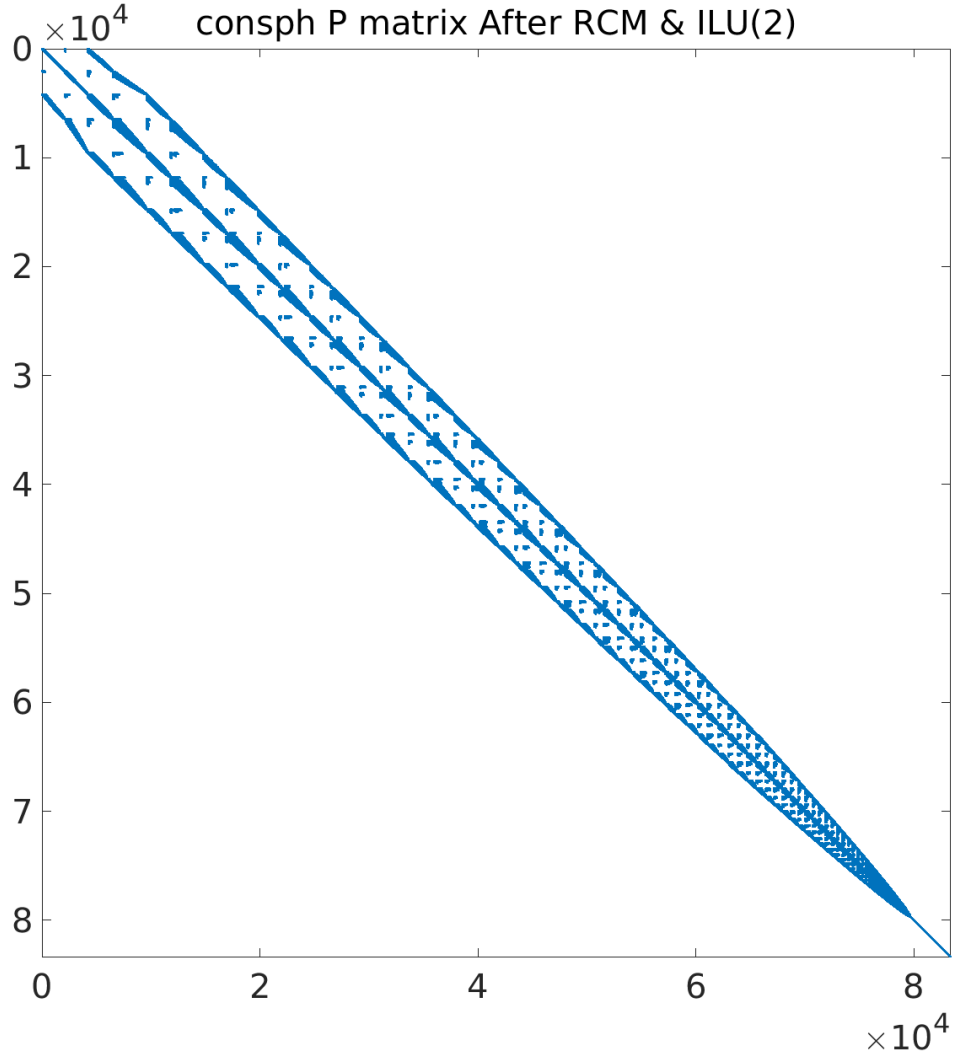

図 4.16: 行列名 consph の RCM と ILU(2) 適用後の非ゼロパターン
(青色: 非ゼロ要素) Nonzeros = 32,175,808 (0.463 %)

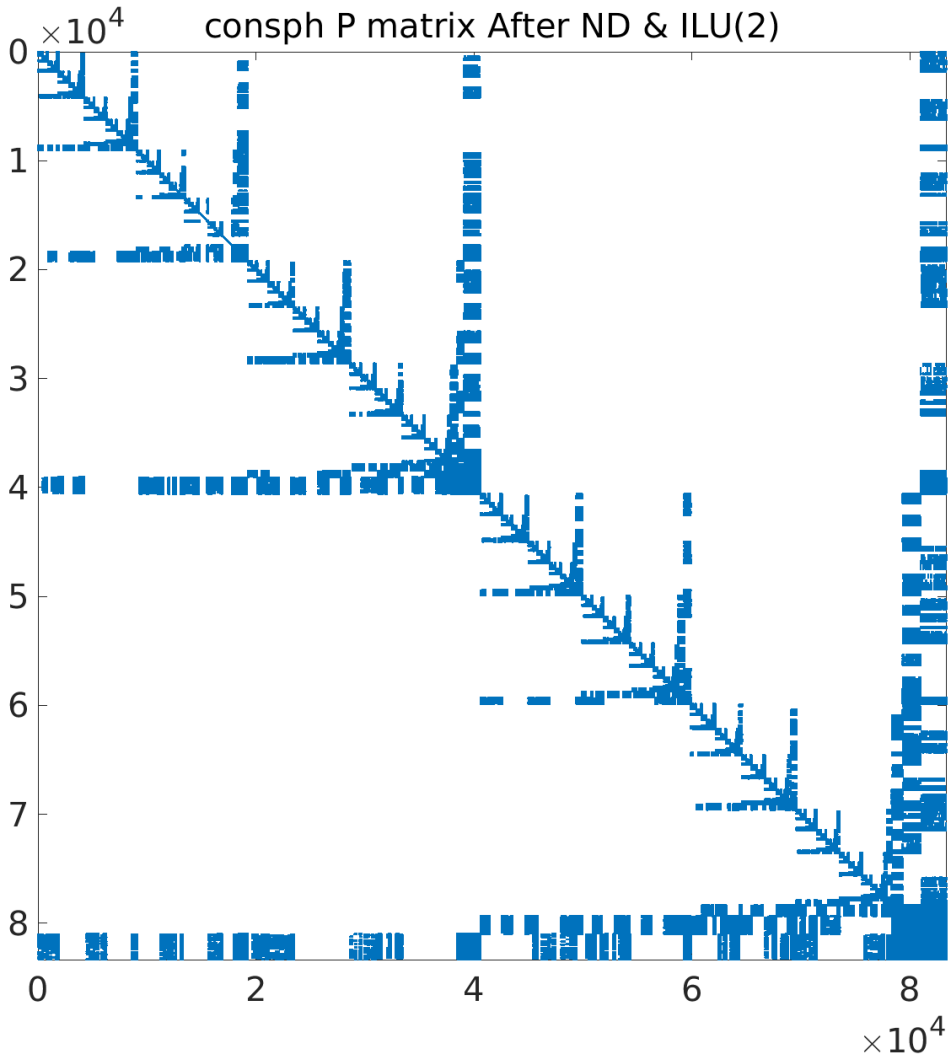

図 4.17: 行列名 consph の ND と ILU(2) 適用後の非ゼロパターン
(青色: 非ゼロ要素) Nonzeros = 42,436,366 (0.611 %)

4.5　係数行列の固有値分布の可視化

図 4.18〜4.22 に 4.3 節で述べた不完全 LU 分解 (ILU) について ILU(0), ILU(1), ILU(2) の組み合わせで、SuiteSparse Matrix Collection の行列名 cant[*6]に適用したときの固有値分布を MATLAB で可視化した結果を示す。

図 4.18〜4.22 からは、ILU 前処理を手厚くする、つまり ILU なし→ ILU(0) → ILU(1) → ILU(2) になるに従って、係数行列 A の固有値分布が原点から離れたところの周りに密集するようになっていることが分かる。

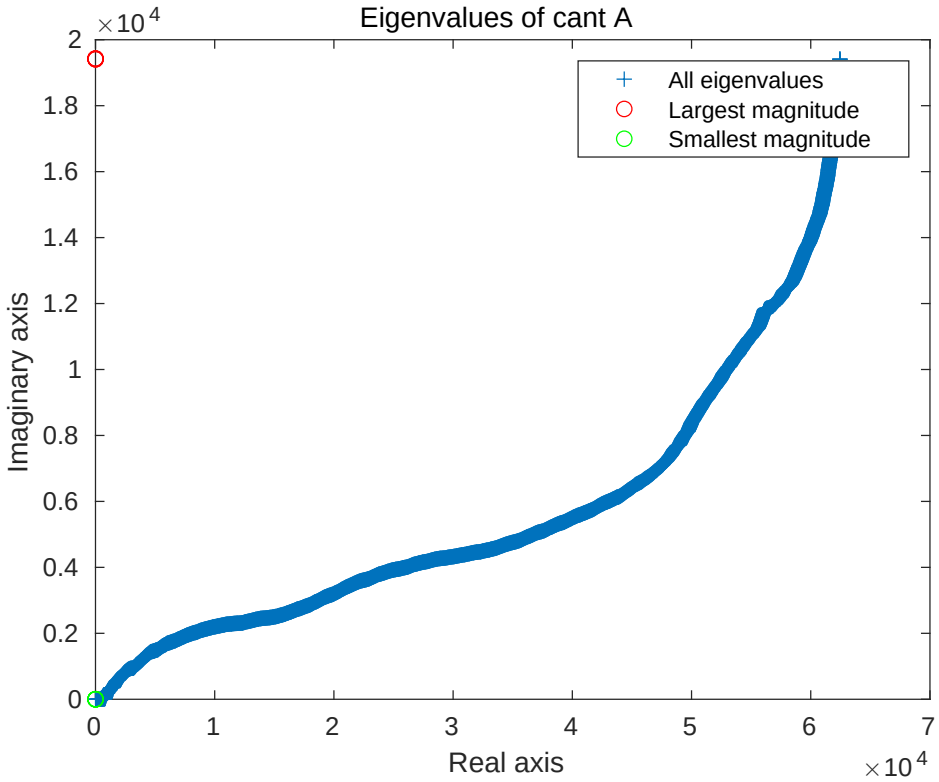

図 4.18: 行列名 cant の全固有値分布

[*6] https://sparse.tamu.edu/Williams/cant

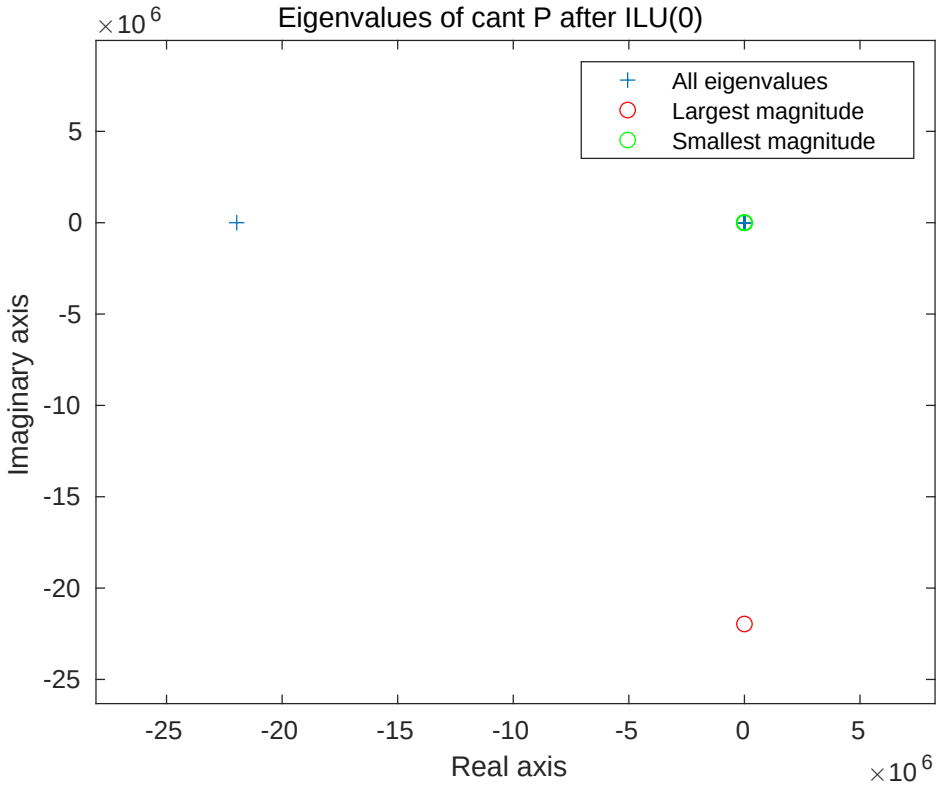

図 4.19: 行列名 cant の ILU(0) 適用後の全固有値分布

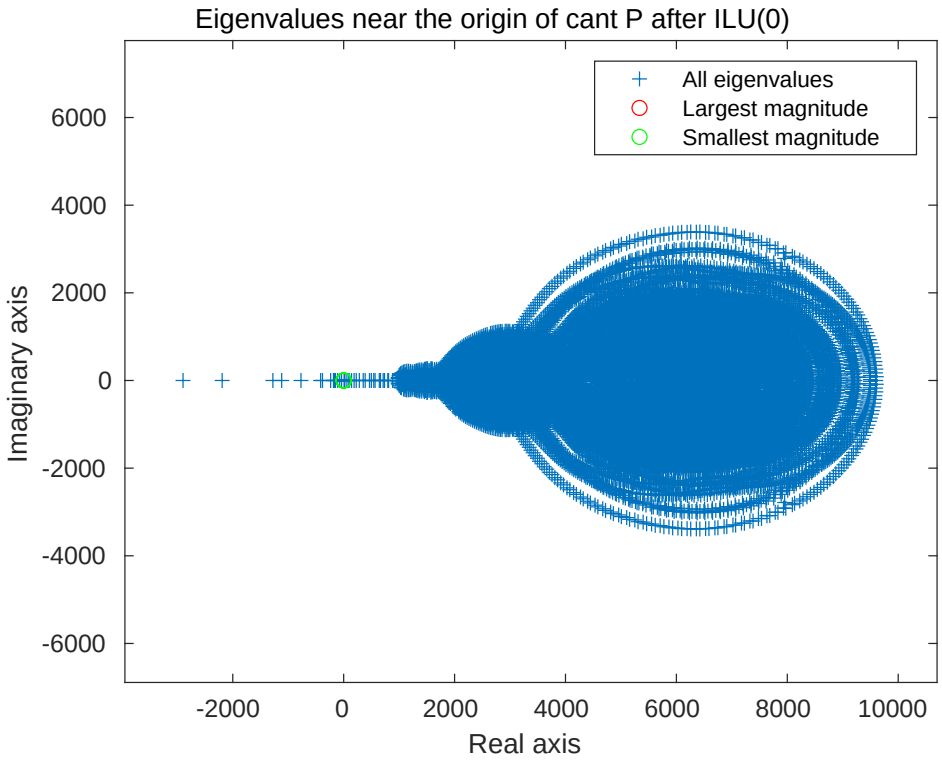

図 4.20: 行列名 cant の ILU(0) 適用後の原点付近の固有値分布

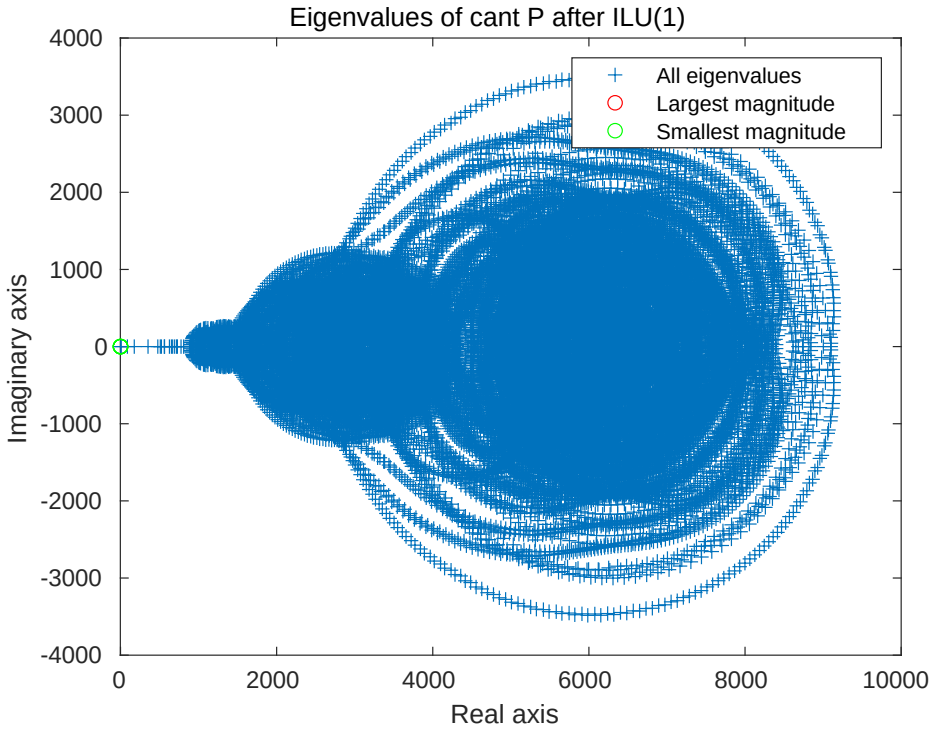

図 4.21: 行列名 cant の ILU(1) 適用後の全固有値分布

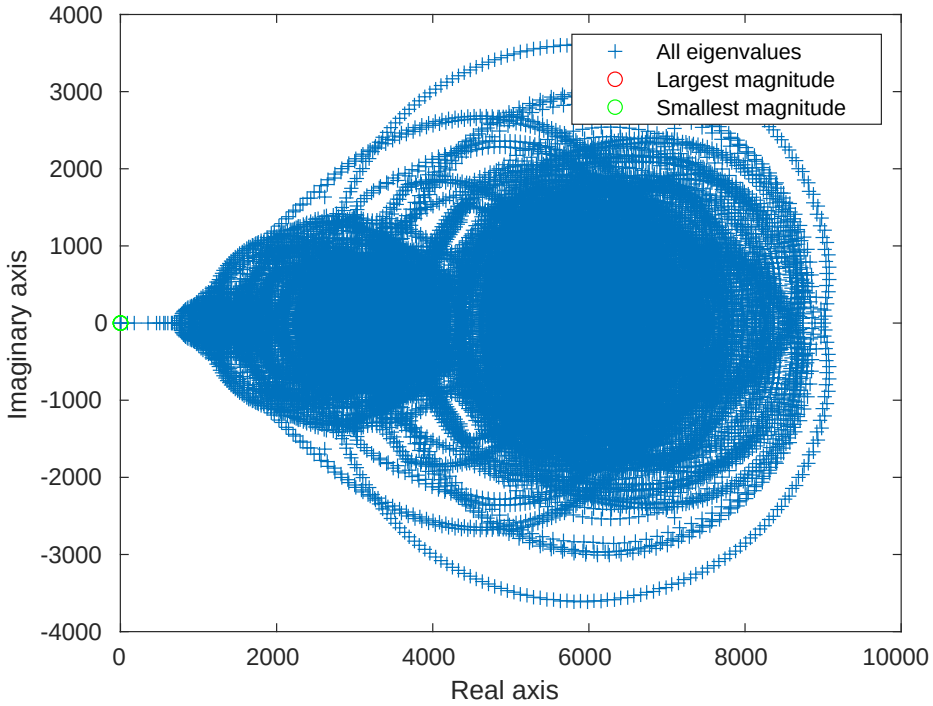

図 4.22: 行列名 cant の ILU(2) 適用後の全固有値分布

4.6　おわりに

本章では、4.2 節でオーダリングについて Cuthill-McKee (CM) と Nested Dissection (ND)、4.3 節で前処理について不完全 LU 分解 (ILU) の議論をした。

不完全 LU 分解についてはフィルインを許すレベルが高いほどクリロフ部分空間法の収束性の向上が期待できるが、フィルインを許しすぎると非ゼロ要素数が増えすぎ、演算量やメモリ使用量が増え処理時間が伸びることが懸念されるという結論に至った。

第5章　実験・評価

5.1　はじめに

本章では第 3 章で議論したグラムシュミット直交化と、第 4 章で議論したオーダリングと前処理について実際にプログラムを用いて評価を行う。

係数行列 A, 右辺ベクトル $\boldsymbol{b}$ には、実利用するシミュレーターから得られた行列 fu_fluid, fu_stru と、実利用するシミュレーターで扱う行列と性質が近いものを SuiteSparse Matrix Collection[*1] [32][33] から選んだ。SuiteSparse Matrix Collection からは行列名 af_shell9[*2]、thermal2[*3]、pwtk[*4]、cant[*5]、parabolic_fem[*6]、F1[*7]、consph[*8]、nd24k[*9] を評価の対象とする。このうち、右辺ベクトル $\boldsymbol{b}$ が含まれていないものについては、解ベクトル $\boldsymbol{x}$ を 0 を含まない乱数とし、$\boldsymbol{b} = A\boldsymbol{x}$ として右辺ベクトル $\boldsymbol{b}$ を生成した。表 5.1 に実験対象の疎行列の概要を示す。表 5.1 の condest は MATLAB で算出した係数行列 A の 1 ノルムの条件数の下限[*10]である。

疎行列格納形式は CSR、演算精度は倍精度浮動小数点で実験を行った。

[*1] https://sparse.tamu.edu/
[*2] https://sparse.tamu.edu/Schenk_AFE/af_shell9
[*3] https://sparse.tamu.edu/Schmid/thermal2
[*4] https://sparse.tamu.edu/Boeing/pwtk
[*5] https://sparse.tamu.edu/Williams/cant
[*6] https://sparse.tamu.edu/Wissgott/parabolic_fem
[*7] https://sparse.tamu.edu/Koutsovasilis/F1
[*8] https://sparse.tamu.edu/Williams/consph
[*9] https://sparse.tamu.edu/ND/nd24k
[*10] https://jp.mathworks.com/help/matlab/ref/condest.html

表 5.1: 実験対象の疎行列 (condest: 係数行列 A の 1 ノルムの条件数の下限)

行列名	行数、列数	非ゼロ要素数	非ゼロ要素率 [%]	condest	種類
af_shell9	504,855	17,588,845	0.00690	1.1975×10^{10}	Subsequent Structural Problem
thermal2	1,228,045	8,580,313	0.00057	7.4806×10^{6}	Thermal Problem
pwtk	217,918	11,524,432	0.02427	5.0348×10^{12}	Structural Problem
cant	62,451	4,007,383	0.10275	5.7616×10^{10}	2D/3D Problem
parabolic_fem	525,825	3,674,625	0.00133	2.1108×10^{5}	Computational Fluid Dynamics Problem
consph	83,334	6,010,480	0.08655	3.1544×10^{7}	2D/3D Problem
F1	343,791	26,837,113	0.02271	3.2618×10^{7}	Structural Problem
fu_stru	130,595	6,953,639	0.04077	5.0949×10^{18}	Structural Problem
nd24k	72,000	28,715,634	0.55393	2.3305×10^{7}	2D/3D Problem
fu_fluid	82,047	2,950,011	0.04382	1.1036×10^{9}	Computational Fluid Dynamics Problem

5.1.1 GPU での実験環境

表 5.2 は GPU での実験環境で、GPU は倍精度浮動小数点演算の性能が高い Tesla シリーズでメモリ容量とメモリバンド幅の大きい A100 [18] を用いた。GPU 版のプログラムは CUDA C/C++を用いて GMRES 本体のみを GPU で実行している。処理時間の計測は `cudaEventRecord()` 関数を用いて行った。

表 5.2: GPU での実験環境

Host	SuperA100
CPU	AMD EPYC 7302 ×2
Memory	DDR4 3200MHz ×16: 256GB
GPU	NVIDIA A100 80GB PCIe
OS	Ubuntu 18.04
CUDA	version 11.5
Compiler Optimization Flags	-O3 -gencode=arch=compute_80,code=sm_80
Library	cuBLAS, cuSPARSE

5.1.2 CPU での実験環境

表 5.3 は CPU での実験環境で、本学の HPC System “KAGAYAKI”[*11]の 1 ノードを用いた。ILU と GMRES のプログラムは C 言語で、オーダリングは MATLAB R2022a で行った。コンパイラ最適化フラグは表 5.3 のものに加え、逐次版は Intel Math Kernel Library (Intel MKL)[*12]における逐次オプション “-qmkl=sequential” を、マルチコア並列版は自動並列化オプション “-parallel” と Intel MKL におけるスレッド並列化オプション “-qmkl=parallel” を付加した。処理時間の計測は `clock_gettime()` 関数を用いて行った。

表 5.3: CPU での実験環境

Host	kagayaki, 1 node
CPU	AMD EPYC 7H12 2.6GHz 64 Cores × 2 Sockets: 128Cores
Memory	DDR4/3200 SDRAM ×16: 512GB
OS	Ubuntu 20.04
Compiler	Intel Compiler (Intel OneAPI 2021)
Compiler Optimization Flags	-O3 -march=core-avx2 -ipo
Library	Intel Math Kernel Library (Intel MKL)

5.2 直交化について GMRES 法の GPU での実験・評価

本節では第 3 章で議論したグラムシュミット直交化について MGS 法と CGS2 法を用いたときの GMRES 法について評価を行う。図 5.1 に GPU でのリスタート付き GMRES 法のフローチャートを示す。

[*11] https://www.jaist.ac.jp/iscenter/mpc/kagayaki/
[*12] https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html

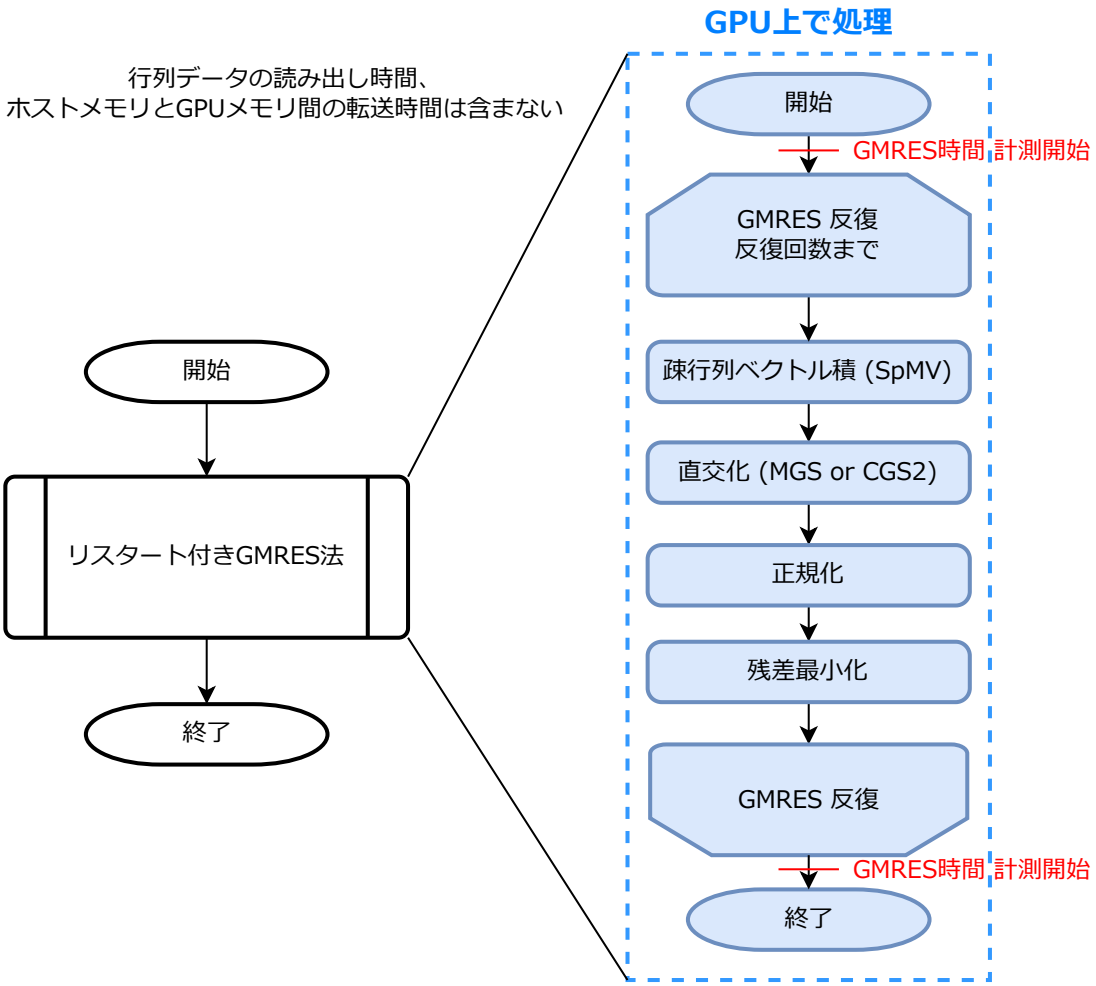

図 5.1: GPU でのリスタート付き GMRES 法のフローチャート

表 5.4 は MGS 法 [2, p.12] と CGS2 法 [29] を用いたときのリスタート付き GMRES 法の処理時間の評価を GPU 環境 (表 5.2) で行ったものである。前処理なし、オーダリングなし、リスタートまでの反復回数は 2000 回、合計反復回数は 4000 回という条件のもとで評価を行った。CGS2 法で使う密行列ベクトル積は `cublasDgemv()` 関数、MGS 法で使う内積は `cublasDdot()` 関数を用いた。なお、行列データの読み出し、ホストメモリと GPU メモリ間のデータ転送 (数ミリ秒〜数十ミリ秒程度) は計測時間に含まれない。

表 5.4: 各直交化手法 (MGS vs CGS2) のリスタート付き GMRES 法の GPU での比較

行列名	直交化	GMRES time [s]	相対残差 $\frac{\|\boldsymbol{r}\|_2}{\|\boldsymbol{b}\|_2}$
F1	MGS	135.659	1.369×10^{-5}
	CGS2	43.851	1.369×10^{-5}
consph	MGS	126.838	6.868×10^{-8}
	CGS2	10.336	6.869×10^{-8}
af_shell9	MGS	140.535	6.831×10^{-6}
	CGS2	60.370	6.722×10^{-6}

表 5.4 から GPU での CGS2 法は MGS 法と比べても収束性は問題なく処理時間が大幅に短いという結果を得た。具体的には、GMRES 処理時間を CGS2 法は MGS 法に比べて最大で 91.9% 短縮した。

5.3 前処理についての実験

本節では第 4 章で議論した前処理について CPU と GPU で実際に前処理付き GMRES 法のプログラムを用いた評価を行う。4.2 節で述べたオーダリングは逆 Cuthill-McKee (RCM) や Nested Dissection (ND) について、4.3 節で述べた不完全 LU 分解は右前処理で ILU(0), ILU(1), ILU(2) について、それぞれの組み合わせでの実験を行った。

5.3.1 オーダリング、ILU 付き GMRES 法の CPU での評価

オーダリング、ILU 前処理付き GMRES 法の CPU (表 5.3) での評価を逐次版と並列版で行った。図 5.2 に CPU での前処理付き GMRES 法のフローチャートを示す。格納方式: CSR, 倍精度, 直交化: CGS2, 最大反復回数: 10000, リスタートなし, 収束判定条件: **相対残差ノルム** (残差ベクトルと右辺ベクトルのノルム比) $\|\boldsymbol{r}\|_2/\|\boldsymbol{b}\|_2$ が 10^{-6} 未満という条件で評価を行った。なお、行列データの読み出しは計測時間に含まれない。

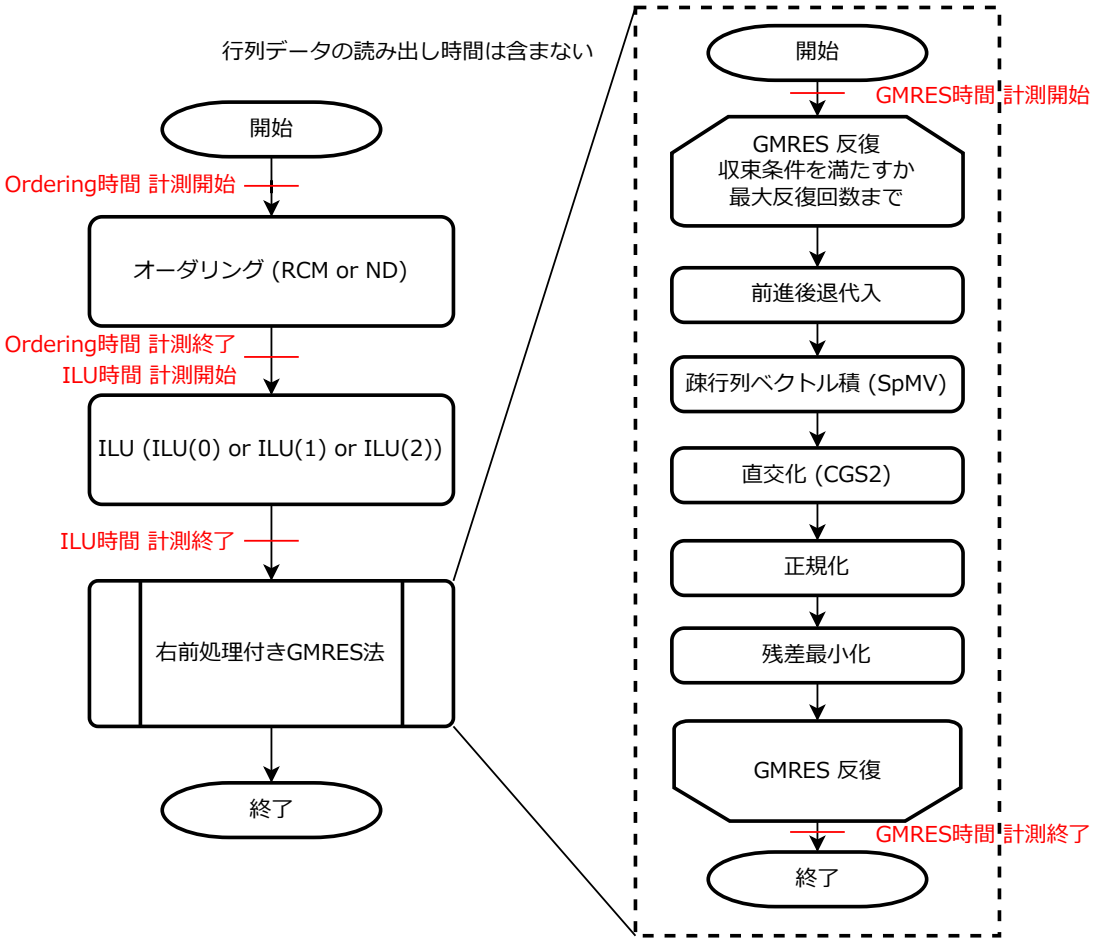

図 5.2: CPU での前処理付き GMRES 法のフローチャート

5.3.1.1 オーダリング、ILU 付き GMRES 法の CPU (逐次版) での評価

表 5.5 にオーダリング、ILU 前処理付き GMRES 逐次版 (1 スレッド) の CPU 環境 (表 5.3) での評価を示す。

表 5.5: オーダリング、ILU 付き GMRES の CPU (逐次版) での評価
(黄色の行: 合計処理時間最短)

行列名	前処理	Ordering	非ゼロ要素数	Ordering 時間 [s]	ILU 時間 [s]	GMRES 時間 [s]	相対残差 $\frac{\|\boldsymbol{r}\|_2}{\|\boldsymbol{b}\|_2}$	反復回数
af_shell9	ILU(0)	None	17,588,845	None	0.6	3,749.3	1.000×10^{-6}	3,268
af_shell9	ILU(0)	RCM	17,588,845	0.5	0.6	2,300.3	9.995×10^{-7}	2,513
af_shell9	ILU(0)	ND	17,588,845	1.5	0.7	5,810.8	9.998×10^{-7}	4,082
af_shell9	ILU(1)	None	25,083,025	None	1.5	831.5	9.993×10^{-7}	1,470
af_shell9	ILU(1)	RCM	22,694,325	0.5	1.3	1,054.9	9.999×10^{-7}	1,674
af_shell9	ILU(1)	ND	30,790,211	1.5	2.7	2,756.3	9.999×10^{-7}	2,718
af_shell9	ILU(2)	None	35,019,925	None	2.7	507.0	9.988×10^{-7}	1,068
af_shell9	ILU(2)	RCM	33,059,675	0.5	2.5	494.8	1.000×10^{-6}	1,063
af_shell9	ILU(2)	ND	48,941,001	1.5	6.3	1,478.4	1.000×10^{-6}	1,888

表は次ページに続く

前ページからの続き

行列名	前処理	Ordering	非ゼロ要素数	Ordering 時間 [s]	ILU 時間 [s]	GMRES 時間 [s]	相対残差 $\frac{\|\boldsymbol{r}\|_2}{\|\boldsymbol{b}\|_2}$	反復回数
thermal2	ILU(0)	None	8,580,313	None	0.4	1,634.6	9.896×10^{-7}	1,347
thermal2	ILU(0)	RCM	8,580,313	0.7	0.2	789.6	9.814×10^{-7}	932
thermal2	ILU(0)	ND	8,580,313	9.1	0.3	1,501.5	9.999×10^{-7}	1,297
thermal2	ILU(1)	None	14,650,303	None	0.9	563.9	9.860×10^{-7}	773
thermal2	ILU(1)	RCM	11,521,539	0.7	0.5	337.0	9.487×10^{-7}	595
thermal2	ILU(1)	ND	13,983,233	9.1	0.8	589.4	9.626×10^{-7}	794
thermal2	ILU(2)	None	22,557,175	None	1.8	253.0	9.920×10^{-7}	497
thermal2	ILU(2)	RCM	16,480,629	0.7	1.0	173.4	9.111×10^{-7}	414
thermal2	ILU(2)	ND	21,210,479	9.1	1.5	263.1	9.130×10^{-7}	513
pwtk	ILU(0)	None	11,524,432	None	0.5	3,575.1	1.000×10^{-6}	5,152
pwtk	ILU(0)	RCM	11,524,432	0.3	0.5	3,507.2	9.980×10^{-7}	5,107
pwtk	ILU(0)	ND	11,524,432	0.9	0.7	N/A	N/A	N/A
pwtk	ILU(1)	None	17,381,576	None	1.6	32.5	9.828×10^{-7}	319
pwtk	ILU(1)	RCM	16,874,900	0.3	1.5	42.7	9.669×10^{-7}	395
pwtk	ILU(1)	ND	21,359,064	0.9	2.7	130.1	9.965×10^{-7}	771
pwtk	ILU(2)	None	26,230,248	None	3.5	19.6	9.987×10^{-7}	180
pwtk	ILU(2)	RCM	24,931,078	0.3	3.1	45.0	9.972×10^{-7}	357
pwtk	ILU(2)	ND	33,826,532	0.9	6.6	20.8	9.789×10^{-7}	166
cant	ILU(0)	None	4,007,383	None	0.2	71.8	9.953×10^{-7}	1,227
cant	ILU(0)	RCM	4,007,383	0.1	0.3	99.7	9.996×10^{-7}	1,472
cant	ILU(0)	ND	4,007,383	1.2	0.5	62.5	9.957×10^{-7}	1,124
cant	ILU(1)	None	8,937,729	None	1.3	5.0	9.770×10^{-7}	144
cant	ILU(1)	RCM	8,799,873	0.1	1.4	4.6	9.248×10^{-7}	135
cant	ILU(1)	ND	10,994,945	1.2	3.5	8.6	9.277×10^{-7}	208
cant	ILU(2)	None	18,095,517	None	5.4	3.9	8.505×10^{-7}	68
cant	ILU(2)	RCM	17,975,967	0.1	5.6	3.7	9.881×10^{-7}	64
cant	ILU(2)	ND	23,510,841	1.2	12.4	5.8	9.484×10^{-7}	90
parabolic_fem	ILU(0)	None	3,674,625	None	0.1	141.4	9.720×10^{-7}	602
parabolic_fem	ILU(0)	RCM	3,674,625	0.2	0.1	29.8	9.668×10^{-7}	260
parabolic_fem	ILU(0)	ND	3,674,625	3.3	0.1	136.3	9.951×10^{-7}	584
parabolic_fem	ILU(1)	None	6,422,527	None	0.2	54.1	9.980×10^{-7}	358
parabolic_fem	ILU(1)	RCM	4,721,153	0.2	0.1	16.1	9.806×10^{-7}	184
parabolic_fem	ILU(1)	ND	6,445,051	3.3	0.4	57.7	9.518×10^{-7}	364
parabolic_fem	ILU(2)	None	10,346,973	None	0.4	25.5	7.969×10^{-7}	227
parabolic_fem	ILU(2)	RCM	6,810,119	0.2	0.1	8.3	9.384×10^{-7}	121
parabolic_fem	ILU(2)	ND	10,255,221	3.3	0.8	29.3	9.130×10^{-7}	240
consph	ILU(0)	None	6,010,480	None	0.3	5,102.7	9.997×10^{-1}	10,000
consph	ILU(0)	RCM	6,010,480	0.2	0.3	5,110.6	9.999×10^{-1}	10,000
consph	ILU(0)	ND	6,010,480	0.7	0.5	5,042.2	1.583×10^{-5}	10,000
consph	ILU(1)	None	13,966,120	None	2.4	5,220.1	9.982×10^{-1}	10,000
consph	ILU(1)	RCM	13,443,806	0.2	2.2	5,303.9	1.000	10,000
consph	ILU(1)	ND	16,887,238	0.7	4.4	5,365.1	1.415×10^{-1}	10,000
consph	ILU(2)	None	34,692,390	None	14.1	26.3	7.274×10^{-7}	257
consph	ILU(2)	RCM	32,175,808	0.2	12.2	5,585.3	1.000	10,000
consph	ILU(2)	ND	42,436,366	0.7	27.0	5,925.7	6.580×10^{-4}	10,000
F1	ILU(0)	None	26,837,113	None	3.5	144.1	9.948×10^{-7}	608
F1	ILU(0)	RCM	26,837,113	0.9	2.2	110.7	9.906×10^{-7}	515
F1	ILU(0)	ND	26,837,113	4.4	2.3	117.5	9.938×10^{-7}	538
F1	ILU(1)	None	171,329,379	None	109.8	93.9	9.963×10^{-7}	216
F1	ILU(1)	RCM	61,975,747	0.9	13.8	56.0	9.995×10^{-7}	234
F1	ILU(1)	ND	67,672,363	4.4	17.9	64.5	9.701×10^{-7}	254
F1	ILU(2)	None	977,782,419	None	4,131.0	122.1	9.890×10^{-7}	80
F1	ILU(2)	RCM	152,264,923	0.9	78.8	44.0	9.160×10^{-7}	110
F1	ILU(2)	ND	165,261,731	4.4	119.5	40.6	9.574×10^{-7}	101
fu_stru	ILU(0)	None	6,953,639	None	0.6	1.6	9.959×10^{-7}	40
fu_stru	ILU(0)	RCM	6,953,639	0.2	0.4	1.3	9.673×10^{-7}	35
fu_stru	ILU(0)	ND	6,953,639	0.8	0.4	1.7	9.852×10^{-7}	46
fu_stru	ILU(1)	None	17,932,747	None	4.6	1.7	6.114×10^{-7}	22
fu_stru	ILU(1)	RCM	13,053,459	0.2	1.6	1.2	7.520×10^{-7}	18
fu_stru	ILU(1)	ND	16,174,247	0.8	2.6	1.8	7.784×10^{-7}	26
fu_stru	ILU(2)	None	51,635,383	None	32.4	2.1	5.842×10^{-7}	13
fu_stru	ILU(2)	RCM	27,792,925	0.2	6.3	1.4	7.526×10^{-7}	12

表は次ページに続く

前ページからの続き

行列名	前処理	Ordering	非ゼロ要素数	Ordering 時間 [s]	ILU 時間 [s]	GMRES 時間 [s]	相対残差 $\frac{\|\boldsymbol{r}\|_2}{\|\boldsymbol{b}\|_2}$	反復回数
fu_stru	ILU(2)	ND	37,116,341	0.8	13.4	2.0	6.626×10^{-7}	16
nd24k	ILU(0)	None	28,715,634	None	10.9	6.7	9.569×10^{-7}	69
nd24k	ILU(0)	RCM	28,715,634	1.1	12.0	0.6	0.000	1
nd24k	ILU(0)	ND	28,715,634	5.7	17.8	1,736.3	8.849×10^{-7}	5,430
nd24k	ILU(1)	None	121,913,942	None	206.7	4.9	9.202×10^{-7}	20
nd24k	ILU(1)	RCM	97,786,230	1.1	140.0	6.0	8.773×10^{-7}	29
nd24k	ILU(1)	ND	133,210,408	5.7	363.8	9.0	8.417×10^{-7}	36
nd24k	ILU(2)	None	486,139,250	None	3,436.8	9.8	9.232×10^{-7}	13
nd24k	ILU(2)	RCM	306,383,182	1.1	1,338.5	7.1	8.426×10^{-7}	13
nd24k	ILU(2)	ND	387,592,020	5.7	2,977.1	9.0	7.655×10^{-7}	14
fu_fluid	ILU(0)	None	2,950,011	None	0.2	699.4	9.742×10^{-7}	3,695
fu_fluid	ILU(0)	RCM	2,950,011	0.1	0.2	5,515.8	2.022×10^{-2}	10,000
fu_fluid	ILU(0)	ND	2,950,011	1.3	0.2	922.1	7.976×10^{-7}	4,330
fu_fluid	ILU(1)	None	8,069,749	None	1.3	648.6	6.925×10^{-7}	3,407
fu_fluid	ILU(1)	RCM	6,246,713	0.1	0.8	5,171.5	5.740×10^{-3}	10,000
fu_fluid	ILU(1)	ND	8,417,819	1.3	1.9	806.3	4.886×10^{-7}	3,824
fu_fluid	ILU(2)	None	21,541,545	None	7.4	836.7	8.157×10^{-7}	3,627
fu_fluid	ILU(2)	RCM	14,423,765	0.1	3.0	5,091.4	5.213×10^{-1}	10,000
fu_fluid	ILU(2)	ND	21,992,685	1.3	11.5	609.2	5.230×10^{-7}	3,019

オーダリングに RCM を用いると反復回数が少なくなって収束性が向上し GMRES の処理時間も短くなっている。概ね ILU(2) と RCM の組み合わせが処理時間が短くなっている。consph は行列は小さいが不安定な問題であり、前処理を手厚くすると収束性が大幅に向上した。

5.3.1.2 オーダリング、ILU 付き GMRES 法の CPU (マルチコア並列版) での評価

表 5.6 にオーダリング、ILU 前処理付き GMRES マルチコア並列版 (128 スレッド並列) の CPU 環境 (表 5.3) での評価を示す。

表 5.6: オーダリング、ILU 付き GMRES 法の CPU (128 スレッド並列) での評価 (黄色の行: 処理時間最短)

行列名	前処理	Ordering	非ゼロ要素数	ILU 時間 [s]	GMRES 時間 [s]	相対残差 $\frac{\|\boldsymbol{r}\|_2}{\|\boldsymbol{b}\|_2}$	反復回数
af_shell9	ILU(0)	None	17,588,845	0.6	4,063.0	1.000×10^{-6}	3,268
af_shell9	ILU(0)	RCM	17,588,845	0.5	2,401.8	9.997×10^{-7}	2,513
af_shell9	ILU(0)	ND	17,588,845	0.7	6,541.6	9.998×10^{-7}	4,082
af_shell9	ILU(1)	None	25,083,025	1.5	922.4	9.993×10^{-7}	1,470
af_shell9	ILU(1)	RCM	22,694,325	1.3	1,157.7	9.997×10^{-7}	1,674
af_shell9	ILU(1)	ND	30,790,211	2.6	2,873.2	9.999×10^{-7}	2,718
af_shell9	ILU(2)	None	35,019,925	2.6	534.6	9.988×10^{-7}	1,068
af_shell9	ILU(2)	RCM	33,059,675	2.4	517.4	1.000×10^{-6}	1,063
af_shell9	ILU(2)	ND	48,941,001	6.1	1,630.5	1.000×10^{-6}	1,888
thermal2	ILU(0)	None	8,580,313	0.4	1,773.4	9.899×10^{-7}	1,347
thermal2	ILU(0)	RCM	8,580,313	0.2	873.0	9.811×10^{-7}	932
thermal2	ILU(0)	ND	8,580,313	0.3	1,637.6	9.980×10^{-7}	1,297
thermal2	ILU(1)	None	14,650,303	0.9	617.6	9.847×10^{-7}	773
thermal2	ILU(1)	RCM	11,521,539	0.5	379.3	9.450×10^{-7}	595
thermal2	ILU(1)	ND	13,983,233	0.8	640.6	9.591×10^{-7}	794
thermal2	ILU(2)	None	22,557,175	1.8	277.3	9.964×10^{-7}	497
thermal2	ILU(2)	RCM	16,480,629	0.9	200.1	9.221×10^{-7}	414
thermal2	ILU(2)	ND	21,210,479	1.5	290.0	9.140×10^{-7}	513

表は次ページに続く

行列名	前処理	Ordering	非ゼロ要素数	ILU 時間 [s]	GMRES 時間 [s]	相対残差 $\frac{\|r\|_2}{\|b\|_2}$	反復回数
pwtk	ILU(0)	None	11,524,432	0.5	4,077.3	9.915×10^{-7}	5,145
pwtk	ILU(0)	RCM	11,524,432	0.5	4,419.9	9.892×10^{-7}	5,108
pwtk	ILU(0)	ND	11,524,432	N/A	N/A	N/A	N/A
pwtk	ILU(1)	None	17,381,576	1.5	40.2	9.828×10^{-7}	319
pwtk	ILU(1)	RCM	16,874,900	1.5	58.7	9.669×10^{-7}	395
pwtk	ILU(1)	ND	21,359,064	2.6	146.6	9.965×10^{-7}	771
pwtk	ILU(2)	None	26,230,248	3.4	30.2	9.987×10^{-7}	180
pwtk	ILU(2)	RCM	24,931,078	3.0	58.1	9.973×10^{-7}	357
pwtk	ILU(2)	ND	33,826,532	6.4	32.9	9.789×10^{-7}	166
cant	ILU(0)	None	4,007,383	0.2	86.6	9.953×10^{-7}	1,227
cant	ILU(0)	RCM	4,007,383	0.2	114.0	9.982×10^{-7}	1,473
cant	ILU(0)	ND	4,007,383	0.5	70.9	9.957×10^{-7}	1,124
cant	ILU(1)	None	8,937,729	1.2	9.7	9.770×10^{-7}	144
cant	ILU(1)	RCM	8,799,873	1.3	6.3	9.248×10^{-7}	135
cant	ILU(1)	ND	10,994,945	3.3	13.4	9.277×10^{-7}	208
cant	ILU(2)	None	18,095,517	5.1	7.8	8.505×10^{-7}	68
cant	ILU(2)	RCM	17,975,967	5.3	7.7	9.881×10^{-7}	64
cant	ILU(2)	ND	23,510,841	11.9	10.4	9.484×10^{-7}	90
parabolic_fem	ILU(0)	None	3,674,625	0.1	164.4	9.720×10^{-7}	602
parabolic_fem	ILU(0)	RCM	3,674,625	0.1	38.8	9.668×10^{-7}	260
parabolic_fem	ILU(0)	ND	3,674,625	0.1	160.0	9.951×10^{-7}	584
parabolic_fem	ILU(1)	None	6,422,527	0.2	64.9	9.980×10^{-7}	358
parabolic_fem	ILU(1)	RCM	4,721,153	0.1	21.5	9.806×10^{-7}	184
parabolic_fem	ILU(1)	ND	6,445,051	0.4	67.9	9.518×10^{-7}	364
parabolic_fem	ILU(2)	None	10,346,973	0.4	34.3	7.969×10^{-7}	227
parabolic_fem	ILU(2)	RCM	6,810,119	0.1	12.2	9.384×10^{-7}	121
parabolic_fem	ILU(2)	ND	10,255,221	0.8	37.2	9.130×10^{-7}	240
consph	ILU(0)	None	6,010,480	0.3	6,703.6	9.995×10^{-1}	10,000
consph	ILU(0)	RCM	6,010,480	0.3	6,392.1	9.999×10^{-1}	10,000
consph	ILU(0)	ND	6,010,480	0.5	6,637.2	1.583×10^{-5}	10,000
consph	ILU(1)	None	13,966,120	2.2	6,486.7	9.866×10^{-1}	10,000
consph	ILU(1)	RCM	13,443,806	2.1	6,634.8	9.999×10^{-1}	10,000
consph	ILU(1)	ND	16,887,238	4.1	6,769.6	1.018×10^{-1}	10,000
consph	ILU(2)	None	34,692,390	13.2	34.5	7.274×10^{-7}	257
consph	ILU(2)	RCM	32,175,808	11.5	7,563.4	1.000	10,000
consph	ILU(2)	ND	42,436,366	25.5	7,191.2	1.048×10^{-3}	10,000
F1	ILU(0)	None	26,837,113	3.4	143.0	9.948×10^{-7}	608
F1	ILU(0)	RCM	26,837,113	2.0	110.6	9.906×10^{-7}	515
F1	ILU(0)	ND	26,837,113	2.2	122.8	9.938×10^{-7}	538
F1	ILU(1)	None	171,329,379	101.1	113.3	9.963×10^{-7}	216
F1	ILU(1)	RCM	61,975,747	13.0	82.8	9.995×10^{-7}	234
F1	ILU(1)	ND	67,672,363	16.9	78.7	9.701×10^{-7}	254
F1	ILU(2)	None	977,782,419	4,220.2	136.5	9.890×10^{-7}	80
F1	ILU(2)	RCM	152,264,923	74.1	51.9	9.160×10^{-7}	110
F1	ILU(2)	ND	165,261,731	112.8	45.1	9.574×10^{-7}	101
fu_stru	ILU(0)	None	6,953,639	0.6	2.3	9.959×10^{-7}	40
fu_stru	ILU(0)	RCM	6,953,639	0.3	1.5	9.673×10^{-7}	35
fu_stru	ILU(0)	ND	6,953,639	0.4	3.2	9.852×10^{-7}	46
fu_stru	ILU(1)	None	17,932,747	4.4	3.6	6.114×10^{-7}	22
fu_stru	ILU(1)	RCM	13,053,459	1.5	2.7	7.520×10^{-7}	18
fu_stru	ILU(1)	ND	16,174,247	2.4	3.8	7.784×10^{-7}	26
fu_stru	ILU(2)	None	51,635,383	30.8	2.6	5.842×10^{-7}	13
fu_stru	ILU(2)	RCM	27,792,925	5.9	2.0	7.526×10^{-7}	12
fu_stru	ILU(2)	ND	37,116,341	12.6	2.8	6.626×10^{-7}	16
nd24k	ILU(0)	None	28,715,634	10.3	8.0	9.569×10^{-7}	69
nd24k	ILU(0)	RCM	28,715,634	11.5	0.9	0.000	1
nd24k	ILU(0)	ND	28,715,634	17.1	2,096.7	8.082×10^{-7}	5,422
nd24k	ILU(1)	None	121,913,942	190.3	6.1	9.202×10^{-7}	20
nd24k	ILU(1)	RCM	97,786,230	131.9	9.2	8.773×10^{-7}	29
nd24k	ILU(1)	ND	133,210,408	349.7	16.1	8.417×10^{-7}	36
nd24k	ILU(2)	None	486,139,250	3,216.5	12.0	9.232×10^{-7}	13
nd24k	ILU(2)	RCM	306,383,182	1,238.8	18.2	8.426×10^{-7}	13
nd24k	ILU(2)	ND	387,592,020	2,864.6	37.0	7.655×10^{-7}	14
fu_fluid	ILU(0)	None	2,950,011	0.2	813.7	9.742×10^{-7}	3,695

表は次ページに続く

行列名	前処理	Ordering	非ゼロ要素数	ILU 時間 [s]	GMRES 時間 [s]	相対残差 $\frac{\|\boldsymbol{r}\|_2}{\|\boldsymbol{b}\|_2}$	反復回数
fu_fluid	ILU(0)	RCM	2,950,011	0.2	6,652.9	1.035×10^{-3}	10,000
fu_fluid	ILU(0)	ND	2,950,011	0.2	1,167.7	7.976×10^{-7}	4,330
fu_fluid	ILU(1)	None	8,069,749	1.3	763.2	6.925×10^{-7}	3,407
fu_fluid	ILU(1)	RCM	6,246,713	0.7	6,761.4	4.280×10^{-3}	10,000
fu_fluid	ILU(1)	ND	8,417,819	1.8	987.1	4.886×10^{-7}	3,824
fu_fluid	ILU(2)	None	21,541,545	6.9	1,099.0	8.178×10^{-7}	3,627
fu_fluid	ILU(2)	RCM	14,423,765	2.7	6,408.6	5.496×10^{-1}	10,000
fu_fluid	ILU(2)	ND	21,992,685	10.9	741.3	5.230×10^{-7}	3,019

CPU での逐次版とマルチコア並列版を比較すると、対象とするアプリケーションはメモリ帯域幅律速であるため、並列化しても速くなるどころか若干遅くなることが確認できた。これはメモリ帯域の飽和と同期オーバーヘッドの影響で性能低下 (反復回数あたりの計算時間が増加) した可能性が挙げられる。

5.3.2 オーダリング、ILU 付き GMRES 法の GPU での評価

図 5.3 に GPU での前処理付き GMRES 法のフローチャートを示す。表 5.7 にオーダリング、ILU 前処理付き GMRES 法の GPU 環境 (表 5.2) での評価を示す。格納方式: CSR, 倍精度, 直交化: CGS2, 最大反復回数: 10000, リスタートなし, 収束判定条件: **相対残差ノルム** (残差ベクトルと右辺ベクトルのノルム比) $\|\boldsymbol{r}\|_2/\|\boldsymbol{b}\|_2$ が 10^{-6} 未満という条件で評価を行った。なお、行列データの読み出し、ホストメモリと GPU メモリ間のデータ転送 (数ミリ秒〜数十ミリ秒程度)、オーダリング、前処理は計測時間に含まれない。

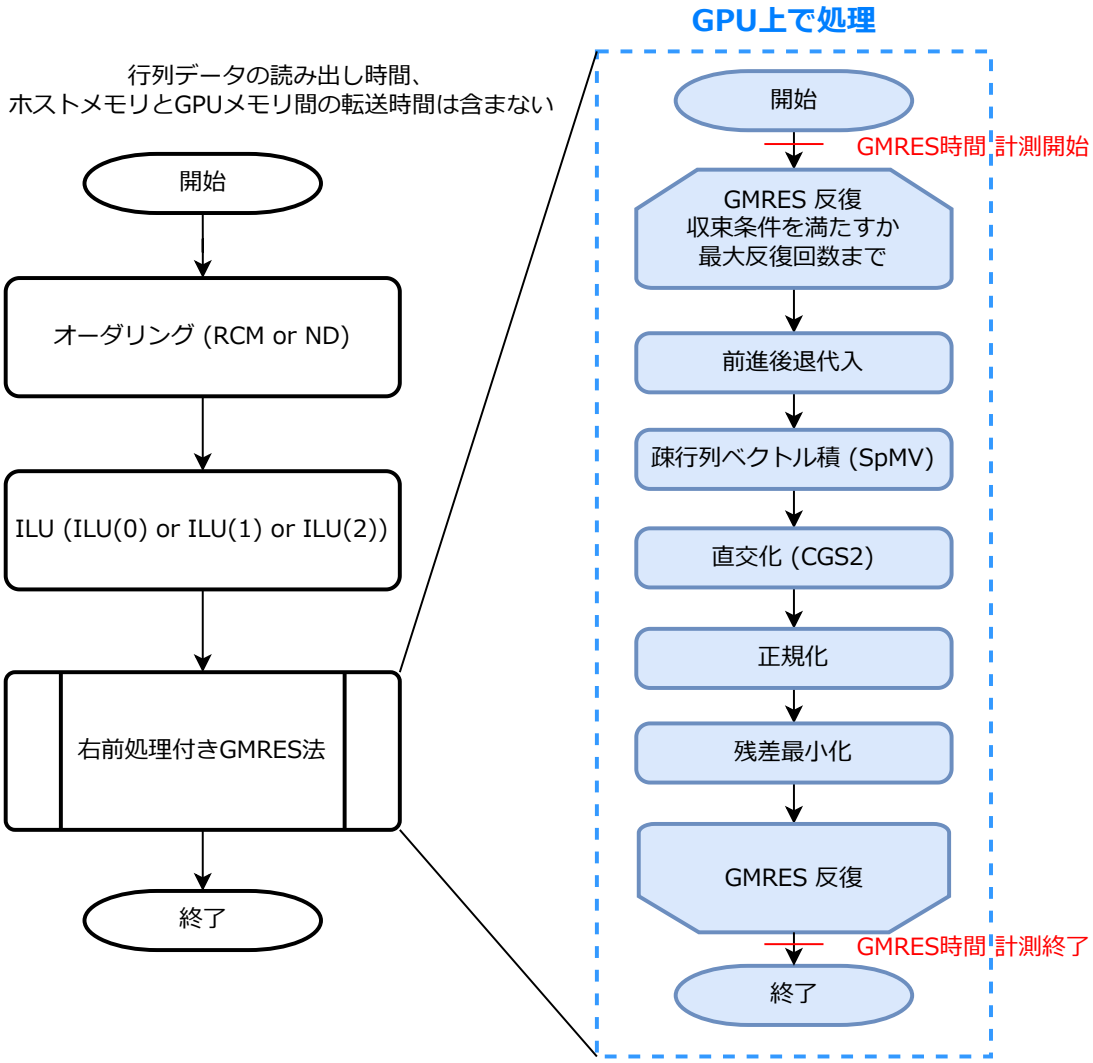

図 5.3: GPU での前処理付き GMRES 法のフローチャート

表 5.7: オーダリング、ILU 付き GMRES 法の GPU での評価
(黄色の行: 処理時間最短)

行列名	前処理	Ordering	非ゼロ要素数	GMRES 時間 [s]	CPU 逐次/GPU GMRES 時間比	相対残差 $\frac{\|r\|_2}{\|b\|_2}$	反復回数
af_shell9	ILU(0)	None	17,588,845	322.5	11.6	8.763×10^{-7}	3,268
af_shell9	ILU(0)	RCM	17,588,845	438.2	5.2	9.997×10^{-7}	2,513
af_shell9	ILU(0)	ND	17,588,845	118.4	49.1	9.998×10^{-7}	4,082
af_shell9	ILU(1)	None	25,083,025	243.6	3.4	9.993×10^{-7}	1,470
af_shell9	ILU(1)	RCM	22,694,325	722.7	1.5	9.997×10^{-7}	1,674
af_shell9	ILU(1)	ND	30,790,211	67.0	41.1	9.999×10^{-7}	2,718
af_shell9	ILU(2)	None	35,019,925	330.4	1.5	9.988×10^{-7}	1,068
af_shell9	ILU(2)	RCM	33,059,675	663.7	0.7	1.000×10^{-6}	1,063
af_shell9	ILU(2)	ND	48,941,001	41.9	35.3	1.000×10^{-6}	1,888
thermal2	ILU(0)	None	8,580,313	83.4	19.6	9.913×10^{-7}	1,887
thermal2	ILU(0)	RCM	8,580,313	248.8	3.2	9.897×10^{-7}	1,145
thermal2	ILU(0)	ND	8,580,313	41.8	35.9	9.879×10^{-7}	1,737
thermal2	ILU(1)	None	14,650,303	81.8	6.9	9.899×10^{-7}	1,055

表は次ページに続く

前ページからの続き

行列名	前処理	Ordering	非ゼロ要素数	GMRES 時間 [s]	CPU 逐次/GPU GMRES 時間比	相対残差 $\frac{\|\boldsymbol{r}\|_2}{\|\boldsymbol{b}\|_2}$	反復回数
thermal2	ILU(1)	RCM	11,521,539	282.5	1.2	9.926×10^{-7}	729
thermal2	ILU(1)	ND	13,983,233	17.0	34.7	9.921×10^{-7}	1,065
thermal2	ILU(2)	None	22,557,175	180.9	1.4	9.900×10^{-7}	691
thermal2	ILU(2)	RCM	16,480,629	333.8	0.5	9.620×10^{-7}	508
thermal2	ILU(2)	ND	21,210,479	8.0	32.8	9.920×10^{-7}	691
pwtk	ILU(0)	None	11,524,432	7,610.7	0.5	2.667×10^{-5}	10,000
pwtk	ILU(0)	RCM	11,524,432	2,750.5	1.3	7.058×10^{-6}	10,000
pwtk	ILU(0)	ND	11,524,432	302.1	N/A	9.966×10^{-6}	10,000
pwtk	ILU(1)	None	17,381,576	278.8	0.1	9.828×10^{-7}	319
pwtk	ILU(1)	RCM	16,874,900	187.8	0.2	9.669×10^{-7}	395
pwtk	ILU(1)	ND	21,359,064	5.0	26.2	9.965×10^{-7}	771
pwtk	ILU(2)	None	26,230,248	155.8	0.1	9.987×10^{-7}	180
pwtk	ILU(2)	RCM	24,931,078	299.9	0.2	9.623×10^{-7}	364
pwtk	ILU(2)	ND	33,826,532	1.5	13.9	9.789×10^{-7}	166
cant	ILU(0)	None	4,007,383	158.0	0.5	9.953×10^{-7}	1,227
cant	ILU(0)	RCM	4,007,383	258.3	0.4	9.995×10^{-7}	1,473
cant	ILU(0)	ND	4,007,383	6.7	9.4	9.958×10^{-7}	1,124
cant	ILU(1)	None	8,937,729	46.6	0.1	9.770×10^{-7}	144
cant	ILU(1)	RCM	8,799,873	51.6	0.1	9.248×10^{-7}	135
cant	ILU(1)	ND	10,994,945	2.7	3.2	9.277×10^{-7}	208
cant	ILU(2)	None	18,095,517	26.9	0.1	8.505×10^{-7}	68
cant	ILU(2)	RCM	17,975,967	25.4	0.1	9.881×10^{-7}	64
cant	ILU(2)	ND	23,510,841	1.7	3.4	9.484×10^{-7}	90
parabolic_fem	ILU(0)	None	3,674,625	2.5	56.4	9.720×10^{-7}	602
parabolic_fem	ILU(0)	RCM	3,674,625	10.4	2.9	9.668×10^{-7}	260
parabolic_fem	ILU(0)	ND	3,674,625	2.5	55.5	9.951×10^{-7}	584
parabolic_fem	ILU(1)	None	6,422,527	4.7	11.4	9.980×10^{-7}	358
parabolic_fem	ILU(1)	RCM	4,721,153	11.9	1.4	9.806×10^{-7}	184
parabolic_fem	ILU(1)	ND	6,445,051	1.2	48.2	9.518×10^{-7}	364
parabolic_fem	ILU(2)	None	10,346,973	4.0	6.4	7.969×10^{-7}	227
parabolic_fem	ILU(2)	RCM	6,810,119	11.4	0.7	9.384×10^{-7}	121
parabolic_fem	ILU(2)	ND	10,255,221	0.7	41.6	9.130×10^{-7}	240
consph	ILU(0)	None	6,010,480	284.6	17.9	1.482×10^{1}	10,000
consph	ILU(0)	RCM	6,010,480	509.4	10.0	4.948	10,000
consph	ILU(0)	ND	6,010,480	120.7	41.8	1.583×10^{-5}	10,000
consph	ILU(1)	None	13,966,120	618.8	8.4	2.155	10,000
consph	ILU(1)	RCM	13,443,806	1,122.9	4.7	1.591×10^{1}	10,000
consph	ILU(1)	ND	16,887,238	301.7	17.8	3.414	10,000
consph	ILU(2)	None	34,692,390	44.9	0.6	7.270×10^{-7}	257
consph	ILU(2)	RCM	32,175,808	3,087.8	1.8	1.346	10,000
consph	ILU(2)	ND	42,436,366	769.7	7.7	2.406×10^{-1}	10,000
F1	ILU(0)	None	26,837,113	12.8	11.3	9.948×10^{-7}	608
F1	ILU(0)	RCM	26,837,113	390.9	0.3	9.906×10^{-7}	515
F1	ILU(0)	ND	26,837,113	4.8	24.3	9.938×10^{-7}	538
F1	ILU(1)	None	171,329,379	26.6	3.5	9.963×10^{-7}	216
F1	ILU(1)	RCM	61,975,747	368.7	0.2	9.995×10^{-7}	234
F1	ILU(1)	ND	67,672,363	9.6	6.7	9.701×10^{-7}	254
F1	ILU(2)	None	977,782,419	41.8	2.9	9.890×10^{-7}	80
F1	ILU(2)	RCM	152,264,923	221.2	0.2	9.160×10^{-7}	110
F1	ILU(2)	ND	165,261,731	9.5	4.3	9.574×10^{-7}	101
fu_stru	ILU(0)	None	6,953,639	0.5	3.2	9.959×10^{-7}	40
fu_stru	ILU(0)	RCM	6,953,639	5.4	0.2	9.673×10^{-7}	35
fu_stru	ILU(0)	ND	6,953,639	0.3	6.2	9.852×10^{-7}	46
fu_stru	ILU(1)	None	17,932,747	2.2	0.8	6.114×10^{-7}	22
fu_stru	ILU(1)	RCM	13,053,459	6.8	0.2	7.520×10^{-7}	18
fu_stru	ILU(1)	ND	16,174,247	0.4	4.1	7.784×10^{-7}	26
fu_stru	ILU(2)	None	51,635,383	4.4	0.5	5.842×10^{-7}	13
fu_stru	ILU(2)	RCM	27,792,925	7.8	0.2	7.526×10^{-7}	12
fu_stru	ILU(2)	ND	37,116,341	0.8	2.4	6.626×10^{-7}	16
nd24k	ILU(0)	None	28,715,634	15.5	0.4	9.569×10^{-7}	69
nd24k	ILU(0)	RCM	28,715,634	4,272.3	0.0	3.949×10^{1}	10,000
nd24k	ILU(0)	ND	28,715,634	2,633.6	0.7	2.183×10^{-1}	10,000
nd24k	ILU(1)	None	121,913,942	6.3	0.8	9.202×10^{-7}	20

表は次ページに続く

前ページからの続き

行列名	前処理	Ordering	非ゼロ要素数	GMRES 時間 [s]	CPU 逐次/GPU GMRES 時間比	相対残差 $\frac{\|r\|_2}{\|b\|_2}$	反復回数
nd24k	ILU(1)	RCM	97,786,230	16.1	0.4	8.773×10^{-7}	29
nd24k	ILU(1)	ND	133,210,408	14.7	0.6	8.417×10^{-7}	36
nd24k	ILU(2)	None	486,139,250	6.6	1.5	9.232×10^{-7}	13
nd24k	ILU(2)	RCM	306,383,182	10.3	0.7	8.426×10^{-7}	13
nd24k	ILU(2)	ND	387,592,020	6.9	1.3	7.655×10^{-7}	14
fu_fluid	ILU(0)	None	2,950,011	38.4	18.2	9.742×10^{-7}	3,695
fu_fluid	ILU(0)	RCM	2,950,011	1,023.0	5.4	2.396×10^{-1}	10,000
fu_fluid	ILU(0)	ND	2,950,011	46.8	19.7	7.558×10^{-7}	4,340
fu_fluid	ILU(1)	None	8,069,749	271.9	2.4	6.964×10^{-7}	3,407
fu_fluid	ILU(1)	RCM	6,246,713	2,270.8	2.3	1.573×10^{-1}	10,000
fu_fluid	ILU(1)	ND	8,417,819	134.6	6.0	4.150×10^{-7}	3,828
fu_fluid	ILU(2)	None	21,541,545	879.1	1.0	8.218×10^{-7}	3,627
fu_fluid	ILU(2)	RCM	14,423,765	4,019.1	1.3	1.991×10^{2}	10,000
fu_fluid	ILU(2)	ND	21,992,685	1,095.5	0.6	5.198×10^{-6}	10,000

オーダリングに RCM を用いると GMRES の処理時間 (プロファイリングによると全体の 9 割近くを占める前進後退代入の処理時間) が大幅に増加している。行列 af_shell9, thermal2, pwtk, cant, parabolic_fem, F1, fu_stru においては、オーダリングに ND を用いると RCM より反復回数が多くなり収束性は RCM より劣るが、反復回数あたりの計算時間が短くなっており、GMRES の処理時間が短縮した。行列 nd24k, fu_fluid についてはオーダリングありだと、オーダリングなしに比べ収束性が悪化し、かつ処理時間も増加した。

consph は行列は小さいが不安定な問題であり、前処理を手厚くすると収束性が大幅に向上した。

5.4 考察

5.4.1 オーダリング、ILU(1) 付き GMRES 法の CPU (逐次版) での性能解析

コード 5.4.1〜5.4.3 に CPU (逐次版) での行列名 F1 に対するオーダリング、ILU(1) 前処理付き GMRES 法について、gprof[*13] によるプロファイリング結果の抜粋 (処理時間上位 5 個) を示す。

以下の関数はそれぞれ

- `ILU_factorize()` 関数: ILU 分解
- `ILU_solve()` 関数: 前進後退代入

を行っている。

[*13] https://sourceware.org/binutils/docs/gprof/

コード 5.4.1: CPU 版 (逐次版) GMRES の行列名: F1, 前処理: ILU(1), オーダリング: None での gprof による性能解析

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 55.98    109.53   109.53        2    54.76    95.48  ILU_factorize
 41.62    190.96    81.43      217     0.38     0.38  ILU_solve
  1.19    193.29     2.33        1     2.33     2.33  CSR2COO
  0.37    194.01     0.72        1     0.72     0.72  Matlab_load
  0.28    194.56     0.55                             compare_column_major
```

コード 5.4.2: CPU 版 (逐次版) GMRES の行列名: F1, 前処理: ILU(1), オーダリング: RCM での gprof による性能解析

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 70.22     39.43    39.43      235     0.17     0.17  ILU_solve
 25.22     53.59    14.16        2     7.08    26.80  ILU_factorize
  1.28     54.31     0.72        1     0.72     0.72  Matlab_load
  1.07     54.91     0.60                             main
  0.80     55.36     0.45        1     0.45     0.45  CSR2COO
```

コード 5.4.3: CPU 版 (逐次版) GMRES の行列名: F1, 前処理: ILU(1), オーダリング: ND での gprof による性能解析

```
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 69.56     46.20    46.20      255     0.18     0.18  ILU_solve
 27.31     64.35    18.14        2     9.07    32.17  ILU_factorize
  1.02     65.03     0.68        1     0.68     0.68  Matlab_load
  0.72     65.51     0.48        1     0.48     0.48  CSR2COO
  0.42     65.79     0.28        1     0.28     0.28  COO2CSR
```

コード 5.4.1〜5.4.3 から、オーダリングを替えると 5 割以上を占める前進後退代入の処理時間がオーダリングなし: 81.43 秒、RCM: 39.43 秒、ND: 46.20 秒と大幅に変化していることが分かる。このことから、CPU では RCM によるオーダリングを行うとデータ依存性が排除され、並列性が向上し、前進後退代入の処理時間が大幅に短くなったと推測できる。

5.4.2 オーダリング、ILU(1) 付き GMRES 法の GPU での性能解析

コード 5.4.4〜5.4.6 に GPU での行列名 F1 に対するオーダリング、ILU(1) 前処理付き GMRES 法について、オーダリングを変えたときの NVIDIA Nsight Systems[*14] による CUDA kernel のプロファイリング結果の抜粋 (処理時間上位 5 個の kernel) を示す。

以下の関数はそれぞれ

- `csrsv2_solve_lower_nontrans_byLevel_kernel()` 関数:
  GPU での疎行列に対する前進代入
- `csrsv2_solve_upper_nontrans_byLevel_kernel()` 関数:
  GPU での疎行列に対する後退代入

を行っている。

コード 5.4.4: GPU 版 GMRES の行列名: F1, 前処理: ILU(1), オーダリング: None での Nsight Systems による性能解析

```
CUDA Kernel Statistics:
Time(%)  Total Time (ns)  Instances  Average (ns)  Name
-------  ---------------  ---------  ------------ ----------------------------------------------------------
   49.8   17,596,700,912        216   81,466,207.9 void csrsv2_solve_upper_nontrans_byLevel_kernel<double,
   48.7   17,186,001,424        216   79,564,821.4 void csrsv2_solve_lower_nontrans_byLevel_kernel<double,
    0.4      141,144,661          1  141,144,661.0 void csrsv2_analysis_upper_nontrans_kernel<(int)5, (int)
    0.4      133,497,051          1  133,497,051.0 void csrsv2_analysis_lower_nontrans_kernel<(int)5, (int)
    0.2       66,973,966        251      266,828.5 std::enable_if<!T7, void>::type internal::gemvx::kernel<
```

コード 5.4.5: GPU 版 GMRES の行列名: F1, 前処理: ILU(1), オーダリング: RCM での Nsight Systems による性能解析

```
CUDA Kernel Statistics:
Time(%) Total Time (ns) Instances  Average (ns)    Name
------- --------------- --------- -------------- ----------------------------------------------------------
   58.7 288,572,913,452       234 1,233,217,578.9 void csrsv2_solve_upper_nontrans_byLevel_kernel<double
   41.1 202,219,458,627       234   864,185,720.6 void csrsv2_solve_lower_nontrans_byLevel_kernel<double
    0.1     397,693,792         1   397,693,792.0 void csrsv2_analysis_upper_nontrans_kernel<(int)5, (in
    0.1     344,646,545         1   344,646,545.0 void csrsv2_analysis_lower_nontrans_kernel<(int)5, (in
    0.0      78,537,341       282       278,501.2 std::enable_if<!T7, void>::type internal::gemvx::kerne
```

[*14] https://developer.nvidia.com/nsight-systems

コード 5.4.6: GPU 版 GMRES の行列名: F1, 前処理: ILU(1), オーダリング: ND での Nsight Systems による性能解析

```
CUDA Kernel Statistics:
Time(%) Total Time (ns) Instances Average (ns)   Name
------- --------------- --------- ------------- ----------------------------------------------------------
   49.9   6,529,067,390       254  25,704,989.7 void csrsv2_solve_lower_nontrans_byLevel_kernel<double,
   42.5   5,562,166,867       254  21,898,294.8 void csrsv2_solve_upper_nontrans_byLevel_kernel<double,
    3.0     393,427,793         1 393,427,793.0 void csrsv2_analysis_upper_nontrans_kernel<(int)5, (int)
    2.1     275,991,149         1 275,991,149.0 void csrsv2_analysis_lower_nontrans_kernel<(int)5, (int)
    0.6      78,790,628       284     277,431.8 std::enable_if<!T7, void>::type internal::gemvx::kernel<
```

コード 5.4.4〜5.4.6 から、オーダリングを替えると 9 割以上を占める前進後退代入の処理時間が大幅に変化していることが分かる。

図 5.4〜5.6 に GPU での行列名 F1 に対するオーダリング、ILU(1) 前処理付き GMRES 法について、オーダリングを変えたときの NVIDIA Nsight Compute[*15] による前進後退代入の CUDA kernel のプロファイリング結果 (メモリーチャート) を示す。

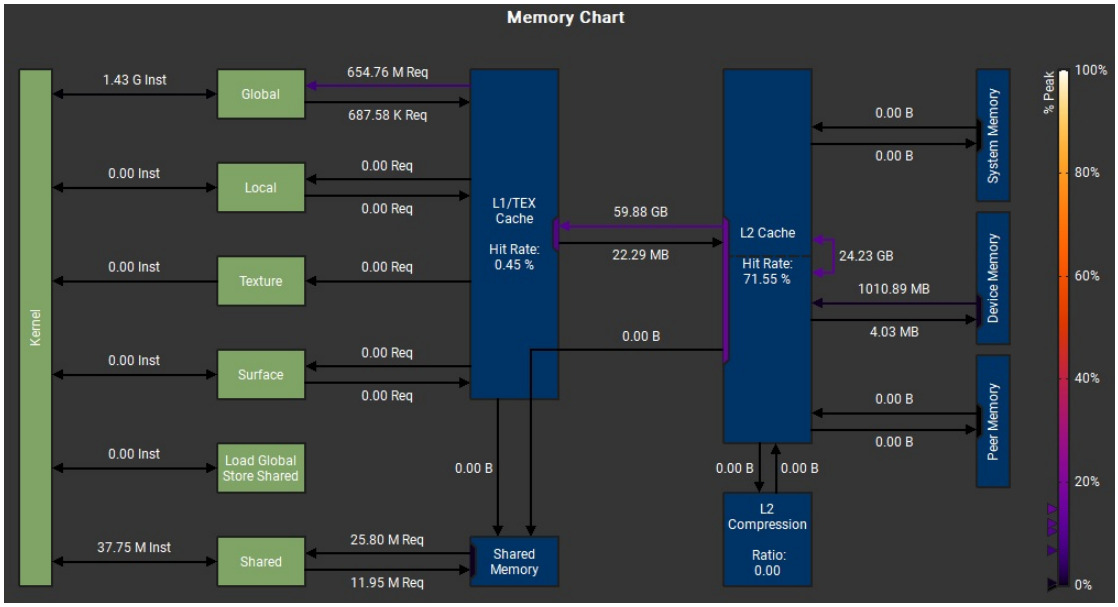

図 5.4: GPU 版 GMRES の行列名: F1, 前処理: ILU(1), オーダリング: None での前進後退代入の Nsight Compute によるメモリーチャート

[*15] https://developer.nvidia.com/nsight-compute

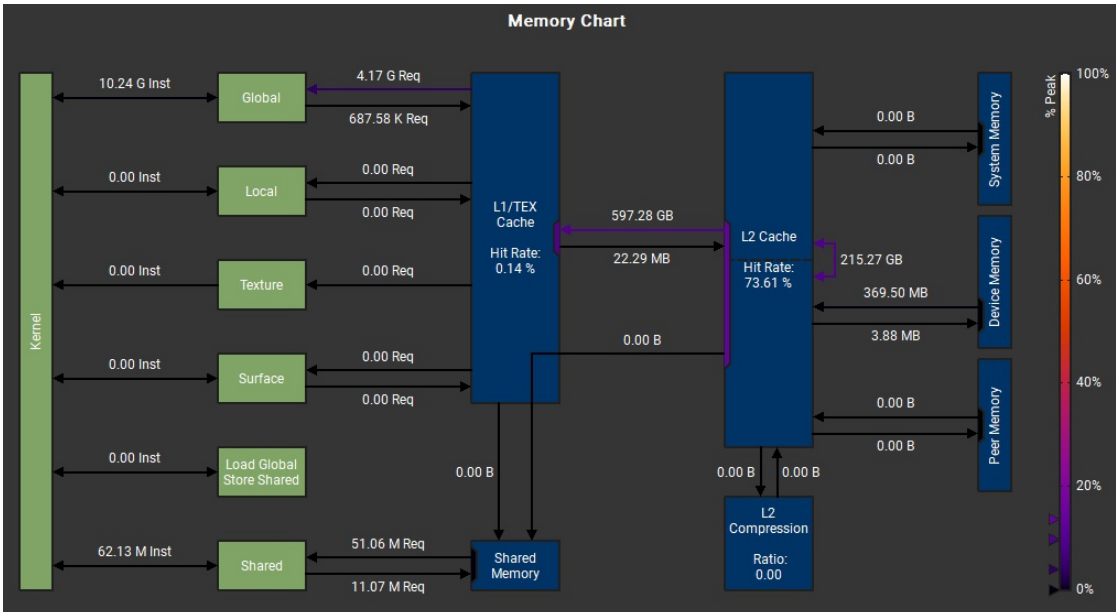

図 5.5: GPU 版 GMRES の行列名: F1, 前処理: ILU(1), オーダリング: RCM での前進後退代入の Nsight Compute によるメモリーチャート

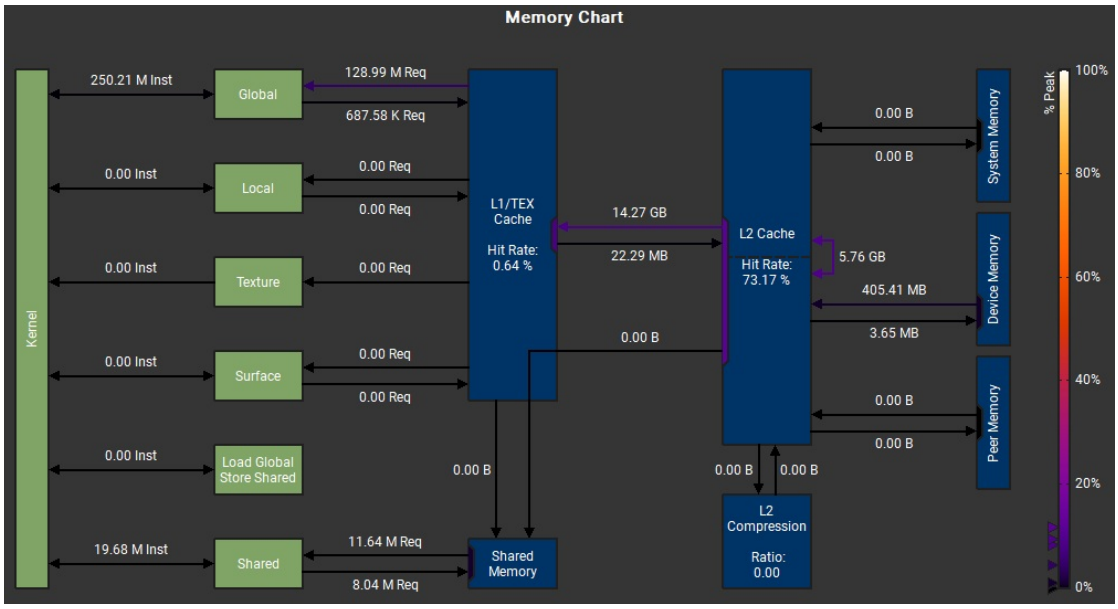

図 5.6: GPU 版 GMRES の行列名: F1, 前処理: ILU(1), オーダリング: ND での前進後退代入の Nsight Compute によるメモリーチャート

図 5.4〜5.6 からオーダリングを変えると、L1 キャッシュヒット率がオーダリングなし: 0.45%、RCM: 0.14%、ND: 0.64%、および L2 キャッシュから L1 キャッシュへのロード量がオーダリングなし: 59.88GB、RCM: 597.28GB、ND: 14.27GB と、大幅に変化していることが分かる。このことから、GPU では ND によるオー

ダリングを行うとメモリアクセス効率が向上し、前進後退代入の処理時間が大幅に短くなったと推測できる。

5.5 おわりに

本章では第3章で議論した直交化、第4章で議論したオーダリング、前処理についてGPUとCPUで実験を行い評価した。直交化についてはCUDAによる評価の結果、GPU上でのCGS2法はMGS法と比べても収束性は問題なく処理時間が大幅に短いということが分かった。前処理についてはILU(0), ILU(1), ILU(2)) 付きのGMRESで評価を行った結果、悪条件の問題の場合は手厚い前処理 (ILU(2) など) が必要になってくることがわかった。また、ILUのフィルインを許すレベルについては、非ゼロ要素数の増加量や収束性などに応じて処理時間が最短となる適切なレベルが異なることがわかった。

オーダリングについてはRCMとNDについてGMRESで評価を行った結果、求解対象の行列によるが概ね、CPUの場合はRCMを適用することで収束性が向上し処理時間が短縮でき、GPUの場合はRCMを適用することで収束性が向上したが処理時間は伸びた。また、GPUの場合はオーダリングにNDを用いると求解対象の行列によるが概ね、GMRESの処理時間を短縮できることが分かった。

第6章　結言

本報告書では、微分方程式を有限差分法や有限要素法によって離散化することで得られる大規模かつ疎な連立一次方程式の GPU に適した数値解法を明らかにすべく、GPU (NVIDIA A100 PCIe) と CPU (HPC System “KAGAYAKI”) で前処理付き GMRES 法の評価を行った。

第 2 章では、連立一次方程式の数値解法と GPU について議論し、研究対象のアプリケーションである疎行列向け反復法 (GMRES 法) はメモリ帯域幅律速であり、kagayaki の CPU と A100 GPU では A100 GPU の方がメモリ帯域幅が広いので、GPU の方が処理性能が出ることが期待できるという結論に達した。

第 3 章では、GMRES 法の直交化で使われる古典グラムシュミット (CGS) 法と修正グラムシュミット (MGS) 法について議論し、計算精度が良い修正グラムシュミット (MGS) 法と、並列性が良い古典グラムシュミット (CGS) 法を 2 回実行する古典グラムシュミット 2 (CGS2) 法ではどちらが処理時間が短いか評価するべきという結論に達した。

第 4 章では、オーダリングについて Cuthill-McKee (CM) と Nested Dissection (ND)、前処理について不完全 LU 分解 (ILU) の議論を行った。不完全 LU 分解についてはフィルインを許すレベルが高いほどクリロフ部分空間法の収束性の向上が期待できるが、フィルインを許しすぎると非ゼロ要素数が増えすぎ、演算量やメモリ使用量が増え処理時間が伸びることが懸念されるという結論に至った。

第 5 章では、第 3 章で議論した直交化について GPU で実験と評価をし、第 4 章で議論した前処理とオーダリングについて CPU と GPU で実験と評価を行った。

直交化については CUDA による評価の結果、GPU 上での CGS2 法は MGS 法と比べても収束性は問題なく処理時間が大幅に短いということが分かった。前処理については ILU(0), ILU(1), ILU(2) 付きの GMRES で評価を行った結果、悪条件の問題の場合は手厚い前処理 (ILU(2) など) が必要になってくることがわかった。また、ILU のフィルインを許すレベルについては、非ゼロ要素数の増加量や収束性などに応じて処理時間が最短となる適切なレベルが異なることがわかった。

オーダリングについては RCM と ND について GMRES で評価を行った結果、求解対象の行列によるが概ね、CPU の場合は RCM を適用することで収束性が向上し処理時間が短縮でき、GPU の場合は RCM を適用することで収束性が向上したが処理時間は伸びた。GPU で RCM 適用するとメモリアクセス効率の低下の影響で性能低下 (反復回数あたりの計算時間増加) した可能性が挙げられる。また、GPU の場合はオーダリングに ND を用いると求解対象の行列によるが概ね、GMRES の

処理時間を短縮できることが分かった。

今後の展望としては、GMRES 法の収束が早くかつ並列性が良いオーダリングの調査・開発や、フィルインの絶対値の大きさによって無視するかどうかを決める ILUT 前処理や ILU よりも強力な前処理 (代数的マルチグリッド (AMG) 法など) を用いた場合の評価などが期待される。

研究業績

- 伊藤 健一, 河野 隆太, 井口 寧. “古典グラムシュミット法と修正グラムシュミット法の GPU 上での比較”, 2021 年度 電気・情報関係学会北陸支部連合大会, F1-5, 1 page, 富山県立大学, Sep. 04, 2021
- 伊藤 健一, 米田 一徳, 岩村 尚, 渡邉 正宏, 井口 寧. “前処理付き GMRES 法の GPU に対する適合性調査”, 2022 年度 電気・情報関係学会北陸支部連合大会, F1-8, 1 page, 金沢大学, Sep. 03, 2022

謝辞

本研究を進めるにあたり、研究方針について助言や指導をしていただいた井口寧教授、河野 隆太助教に感謝いたします。

また、共同研究においてご協力やプログラムのご提供、有益な助言を頂きました富士通 Japan 株式会社の渡邉 正宏様、岩村 尚様、米田 一徳様にお礼申し上げます。

本研究成果の一部は、本学情報社会基盤研究センターの HPC System “KAGAYAKI” を利用して得られたものです。

参考文献

[1] 日本応用数理学会, 櫻井鉄也, 松尾宇泰, and 片桐孝洋. “数値線形代数の数理と HPC (シリーズ応用数理 第 6 巻).” 単行本. 共立出版, Aug. 30, 2018, p. 336. ISBN: 978-4320019553.

[2] Yousef Saad. “Iterative Methods for Sparse Linear Systems (2nd edition).” 2003. DOI: 10.1137/1.9780898718003.ch4. URL: https://www-users.cse.umn.edu/~saad/IterMethBook_2ndEd.pdf.

[3] Nathan Bell and Michael Garland. “Efficient Sparse Matrix-Vector Multiplication on CUDA.” NVIDIA Technical Report no. NVR-2008-004. NVIDIA Corporation, Dec. 2008. URL: https://research.nvidia.com/publication/2008-12_efficient-sparse-matrix-vector-multiplication-cuda.

[4] Georgii Evtushenko. “Sparse Matrix-Vector Multiplication with CUDA | by Georgii Evtushenko | Analytics Vidhya | Medium.” Nov. 2019. URL: https://medium.com/analytics-vidhya/sparse-matrix-vector-multiplication-with-cuda-42d191878e8f.

[5] David Patterson and John Hennessy. “コンピュータの構成と設計 MIPS Editoin 第 6 版 下.” Trans. by 成田 光彰. 単行本. 日経 BP, Nov. 3, 2021, p. 408. ISBN: 978-4296070107.

[6] 河村 知記 and 井口 寧. “GPGPU による超大規模連立一次方程式の求解高速化に向けた省メモリ指向疎行列格納方式に関する研究.” PhD thesis. Sept. 2020. URL: http://hdl.handle.net/10119/17001.

[7] 山本有作. “計算科学技術特論 A (2021) 第 8 回 配信講義 行列計算における高速アルゴリズム 1.” URL: https://www.r-ccs.riken.jp/about/careers/e-learning/hpc-4-comp-sci-2021a/tokurona-2021-8/.

[8] MathWorks. “スパース行列の演算 - MATLAB & Simulink - MathWorks 日本.” URL: https://jp.mathworks.com/help/matlab/math/sparse-matrix-operations.html.

[9] 渡部善隆. “いつ反復計算をやめるべきか? : 収束判定基準の設定方法.” 九州大学大型計算機センター広報, vol. 32. no. 1 (Mar. 1999), pp. 11–30. ISSN: 0389-7885. DOI: info:doi/10.15017/1470342.

[10] 中島研吾. “有限要素法による一次元定常熱伝導解析プログラム C 言語編.” 東京大学情報基盤センター. URL: http://nkl.cc.u-tokyo.ac.jp/FEM/02-FEM1D-C.pdf.

[11] Youcef Saad and Martin H. Schultz. “GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems.” SIAM Journal on Scientific and Statistical Computing, vol. 7. no. 3 (1986), pp. 856–869. DOI: 10.1137/0907058.

[12] 山本 有作. “エクサフロップス時代に向けた線形計算アルゴリズムの課題と研究動向 (応用数理と計算科学における理論と応用の融合).” 数理解析研究所講究録, no. 2005 (Nov. 2016), pp. 32–42. ISSN: 1880-2818. URL: https://ci.nii.ac.jp/naid/120006477688/.

[13] 杉原正顯 and 室田一雄. “線形計算の数理.” ペーパーバック. 岩波書店, Dec. 13, 2016, p. 394. ISBN: 978-4007305504.

[14] 仁木滉 and 河野敏行. “楽しい反復法.” 単行本. 共立出版, Jan. 1, 1998, p. 128. ISBN: 978-4320015821.

[15] ストラング ギルバート. “世界標準 MIT 教科書 ストラング: 線形代数イントロダクション.” Trans. by 松崎公紀 and 新妻弘. 単行本. 近代科学社, Dec. 22, 2015, p. 608. ISBN: 978-4764904057.

[16] Richard Barrett, Michael W. Berry, Tony F. Chan, James Demmel, June Donato, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Charles Romine, and Henk A. van der Vorst. “4. Related Issues.” *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, pp. 57–81. DOI: 10.1137/1.9781611971538.ch4. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611971538.ch4.

[17] NVIDIA Corporation. “Programming Guide :: CUDA Toolkit Documentation.” eprint: https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[18] NVIDIA Corporation. “NVIDIA A100 Tensor Core GPU.” Tech. rep. 2020. URL: https://www.nvidia.com/content/dam/en-zz/ja/Solutions/Data-Center/documents/nvidia-ampere-architecture-whitepaper-jp.pdf.

[19] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: An Insightful Visual Performance Model for Multicore Architectures.” Commun. ACM, vol. 52. no. 4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785.

[20] A U.S. Department of Energy National Laboratory. “Performance and Algorithms Research » Research » Roofline » Introduction.” URL: https://crd.lbl.gov/divisions/amcr/computer-science-amcr/par/research/roofline/introduction/.

[21] Naruse Akira. “VOLTA AND TURING: ARCHITECTURE AND PERFORMANCE OPTIMIZATION.” NVIDIA, Sept. 2018. URL: https://www.nvidia.com/content/apac/gtc/ja/pdf/2018/2051.pdf.

[22] NVIDIA Corporation. “NVIDIA TESLA V100 GPU アーキテクチャ.” Tech. rep. 2017. URL: https://images.nvidia.com/content/pdf/tesla/Volta-Architecture-Whitepaper-v1.1-jp.pdf.

[23] 石上裕之, 木村欣司, and 中村佳正. “古典グラム・シュミット法に基づく逐次直交化計算の並列実装について.” 情報処理学会研究報告. [ハイパフォーマンスコンピューティング], vol. 2014-HPC-145. no. 20 (July 2014), pp. 1–6. URL: https://ci.nii.ac.jp/naid/110009808115/.

[24] 中島 研吾. “マルチコア時代の並列前処理手法.” 数理解析研究所講究録, vol. 1733 (2011), pp. 1–10. URL: https://www.kurims.kyoto-u.ac.jp/~kyodo/kokyuroku/contents/pdf/1733-01.pdf.

[25] NVIDIA Corporation. “Incomplete-LU and Cholesky Preconditioned Iterative Methods Using cuSPARSE and cuBLAS.” Tech. rep. 2011. eprint: https://docs.nvidia.com/cuda/pdf/Incomplete_LU_Cholesky.pdf. URL: https://docs.nvidia.com/cuda/incomplete-lu-cholesky/index.html.

[26] Hartwig Anzt, Edmond Chow, and Jack Dongarra. “ParILUT—A New Parallel Threshold ILU Factorization.” SIAM Journal on Scientific Computing, vol. 40. no. 4 (2018), pp. C503–C519. DOI: 10.1137/16M1079506.

[27] Hartwig Anzt, Tobias Ribizel, Goran Flegar, Edmond Chow, and Jack Dongarra. “ParILUT - A Parallel Threshold ILU for GPUs.” *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2019, pp. 231–241. DOI: 10.1109/IPDPS.2019.00033.

[28] Azzam Haidar, Harun Bayraktar, Stanimire Tomov, Jack Dongarra, and Nicholas J. Higham. “Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems.” Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences, vol. 476. no. 2243 (2020), p. 20200110. DOI: 10.1098/rspa.2020.0110.

[29] L. Giraud, J. Langou, M. Rozložník, and J. D. Eshof. “Rounding error analysis of the classical Gram-Schmidt orthogonalization process.” Numerische Mathematik, vol. 101 (2005), pp. 87–100. DOI: 10.1007/s00211-005-0615-4.

[30] 中島研吾. “科学技術計算のための マルチコアプログラミング入門 第III部：オーダリング.” 東京大学情報基盤センター. URL: http://nkl.cc.u-tokyo.ac.jp/seminars/multicore/omp-c-03.pdf.

[31] E. Cuthill and J. McKee. “Reducing the Bandwidth of Sparse Symmetric Matrices.” *Proceedings of the 1969 24th National Conference*. ACM ’69. New York, NY, USA: Association for Computing Machinery, 1969, pp. 157–172. ISBN: 9781450374934. DOI: 10.1145/800195.805928.

[32] Timothy A. Davis and Yifan Hu. “The University of Florida Sparse Matrix Collection.” ACM Trans. Math. Softw. vol. 38. no. 1 (Dec. 2011). ISSN: 0098-3500. DOI: 10.1145/2049662.2049663.

[33] Scott P. Kolodziej, Mohsen Aznaveh, Matthew Bullock, Jarrett David, Timothy A. Davis, Matthew Henderson, Yifan Hu, and Read Sandstrom. “The SuiteSparse Matrix Collection Website Interface.” Journal of Open Source Software, vol. 4. no. 35 (2019), p. 1244. DOI: 10.21105/joss.01244.

[34] George Karypis and Vipin Kumar. “A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs.” SIAM Journal on Scientific Computing, vol. 20. no. 1 (1998), pp. 359–392. DOI: 10.1137/S1064827595287997.

[35] Maxim Naumov. “Parallel Solution of Sparse Triangular Linear Systems in the Preconditioned Iterative Methods on the GPU.” NVIDIA Technical Report no. NVR-2011-001. NVIDIA Corporation, June 2011. URL: https://research.nvidia.com/publication/2011-06_parallel-solution-sparse-triangular-linear-systems-preconditioned-iterative.

索 引