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

北陸先端科学技術大学院大学

河村　知記

博 士 論 文

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

河村　知記

主指導教員　井口　寧

北陸先端科学技術大学院大学

情報科学研究科

令和2年9月

概 要

近年，自組織内の小規模な計算資源を使用するオンサイト環境での大規模かつ高精度な数値シミュレーションの需要が拡大している．数値シミュレーションで度々用いられる Finite Different Method (FDM) や Finite Element Method (FEM) は，最終的に大規模な連立一次方程式を解く必要があり，膨大な計算量となる．そのためオンライン環境のような限定された環境において，数値シミュレーションを実用化するためには，小規模ながら高い演算性能を持つ計算機が必要である．そこで小規模ながら高い演算性能を持つ GPU を活用し，オンサイト環境において数値シミュレーションを実用的な時間で行う試みが多く行われている．GPU は CPU に比べ非常に多くの演算コアを搭載しており，GPU を用いることで多くのアプリケーションにおいて高速化が可能である．しかしながら GPU を用いた演算を行うためにはアプリケーションで使用するデータを予め GPU メモリ上へ転送する必要があるが，この GPU メモリが CPU メモリに比べ非常に少量であるという問題点がある．連立一次方程式の求解手法は多く存在するが，ほぼ全ての手法において連立一次方程式を表す疎行列を格納するために莫大なメモリ領域が必要である．この疎行列を GPU 上のメモリへ格納しきれない場合には，CPU と GPU 間で複数回データ転送を行う必要があり，多くの時間を費やす．そこで本研究では，FDM や FEM で生成される疎行列の規則性を考慮した省メモリ指向疎行列格納方式，Pattern Compression (PatComp) 法と Row Block Packing (RBP) 法を提案する．PatComp 法は疎行列中の複数の行において非ゼロ要素の並びが一致していることに着目し，非ゼロ要素の位置関係をパターンを用いて表すことでメモリ使用量の削減を行う手法である．疎行列中の各行の非ゼロ要素の並びをパターン化し，パターンをテーブルに登録することで，少ないパターンから全ての行の非ゼロ要素の位置関係を復元することが可能である．そのため PatComp のメモリ使用量は既存手法に比べ非常に少なくなる．RBP 法は，PatComp の変換時間が長いことから使用可能な問題が限定されるという問題点を解消するため，変換処理を高速化した手法である．疎行列内に多くの連続した非ゼロ要素が存在することに着目し，それらの位置を記憶するために必要となる情報量を削減する．RBP 法と PatComp 法のメモリ削減率の評価を行った結果，PatComp 法は 15 個中 4 個の疎行列において 30.0%以上，12 個の疎行列において 25%以上のメモリ使用量削減を達成した．RBP 法を用いることで既存疎行列格納方式のメモリ使用量を最大 28.0%削減することに成功した．SpMV 演算性能に関する評価では RBP 法，PatComp 法を用いても演算性能は既存の疎行列を用いた場合と遜色ないことを明らかにした．また最後に実際の数値シミュレーションで RBP 法，PatComp 法を使用することを想定し，GMRES を用いた演算時間評価実験を行った．その結果，RBP 法を用いた GMRES の演算時間は変換に要する時間を削減し，既存疎行列格納方式を用いた場合に比べ，平均 3.1%の短くなった．本研究で提案した RBP 法，PatComp 法は数値シミュレーションで使用するメモリ使用量を大幅に削減し，さらに大規模かつ高精度な数値シミュレーションをオンサイト環境で行うことを可能にした．

目 次

図 目 次

表 目 次

List of Algorithms

第1章　緒言

近年，医療分野をはじめとする多くの分野において高精度な数値シミュレーションが必要とされている．そして数値シミュレーションに求められる規模や精度は年々上昇しており，数値シミュレーションの計算量は大幅に増加している．一般的に高精度な数値シミュレーションには，PC クラスタやスーパーコンピュータ等の大規模な計算機を使用することが多い．しかし，このような大規模な計算機は導入や管理のコストが高いため，外部組織が所有するものを借りてシミュレーションを行うケースも頻繁に見受けられる．

一例として医療分野における数値シミュレーション応用を挙げる．この分野では Heartflow 社が $\mathrm{FFR}_{\mathbf{CT}}$[1] の商用サービスが始まっており，これは Amazon Web Services(AWS) を活用したクラウドサービスである [2]．このサービスでは，狭心症が疑われる患者の CT 画像と診断情報を病院から AWS 上に構築されたサービス基盤に送信する．その後，Heartflow の技術者が Finite Element Method (FEM) による 3 次元の流体解析のより得られた Fractional Flow Reserve(FFR) 値を，病院にレポートとして返すというものである．FFR 値とは，冠動脈内の狭窄によりどの程度血流が阻害されるかを表す値であり，薬剤により心臓の冠動脈の末梢を拡張させた状態で，狭窄部の下流の血圧を上流の血圧で割ることで算出される．これまで実測の FFR 値を求めるには，カテーテルを用いた侵襲的な検査が必要だったが，FFRct のサービスでは数値シミュレーションを活用して非侵襲的に FFR 値を算出することができる．一方で，情報管理の強化や医師がシミュレーションをコントロールしたいという要望もあるため，オンサイト環境のシミュレーション実行が望まれるケースがある．オンライン環境とは，自組織内の計算資源のみを使用する環境のことである．これに対しては，院内に配置可能なサイズのワークステーションで現実的な時間内に解析できるよう，次元削減した数理モデルを用いて FFR 値を算出する研究が行われており [3][4]，数十分での解析が実現されている．しかしながら，次元削減により流速や血圧の空間分布あるいはずり応力といった様々な詳細な情報が抜けているという欠点があり，オンサイト環境での実行が可能でありながら，より高精度なシミュレーションが求められている．

上記のようなオンサイト環境での高精度な数値シミュレーション実現のために，General-Purpose computing on Graphic Processing Unit (GPGPU) の活用が考えられる．GPGPU は，従来描画処理に使用されていた Graphic Processing Unit (GPU) を汎用的なアプリケーションの並列化に使用し，高速化を図る技術である．GPU は非常に多くの演算コアを備えていることから，GPU を用いて様々な高い並列度を有したアプリケーションの高速化が可能である [5][6][7]．GPGPU の発展により，従来大規模計算機を必要としていた

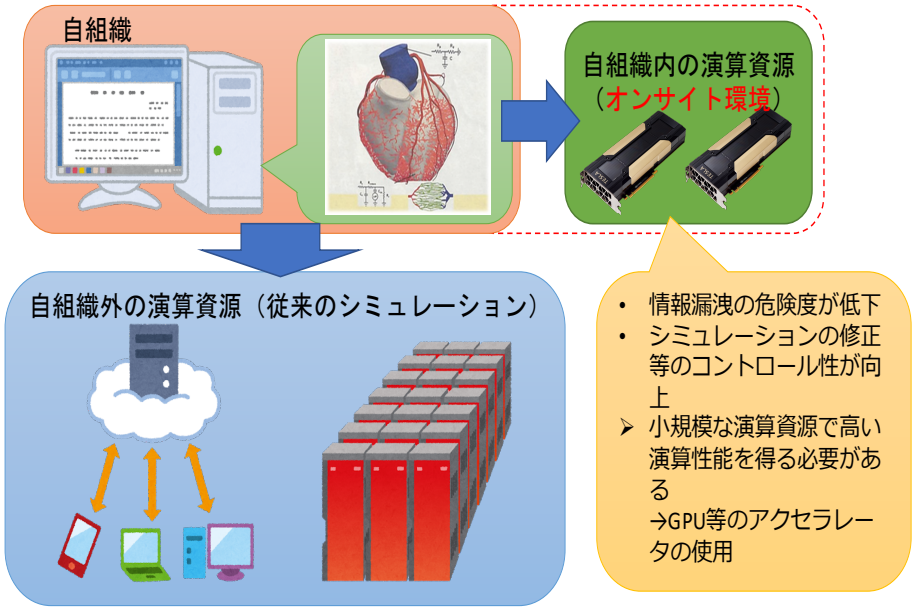

図 1.1: The simution on on-site

高精度な数値シミュレーションを実用的な時間，かつ省スペースで実行可能となった．しかしながら，利用可能な GPU メモリ量の制約がある．CPU は 1 ノードで数 TB まで利用可能なのに対して，GPGPU では高々数十 GB しか利用できない．数値シミュレーションの代表的な手法である Finite Element Method (FEM) や Finite Different Method(FDM) 等を用いた場合，メッシュ数の増加によるシミュレーションの大規模化やメッシュの細分化によるシミュレーションの高精度化によって解析で扱う疎行列のサイズは増大する．GPGPU を使用して数値シミュレーションを解く際には，CPU メモリに格納されている疎行列データを GPU メモリへ転送し，GPU 上で演算する必要がある．この疎行列が GPU メモリへ格納しきれない場合，GPU 上のデータを CPU メモリへ退避させる必要があり，CPU と GPU 間の転送が頻発することからデータ転送に多くの時間を費やすこととなる．

このことから数値シミュレーションで扱う疎行列の省メモリ化手法を確立し，GPGPU のボトルネックを解消することが，オンサイト環境における数値シミュレーションのさらなる発展に必須である．疎行列の省メモリ化により，シミュレーション範囲の拡大，精度の向上，同時に実行できるシミュレーション数の増加等，様々な利点が存在する．例えば先に挙げた心臓シミュレーションにおいては，同時に実行可能な検査数の増加やミュレーション精度の向上による検査時間の短縮，偽陰性の削減等の利点が考えられ，患者の負担が大きく低下する．また少容量メモリを搭載されている安価な GPU や Jetson のような組み込み用 GPU においても数値シミュレーションが利用できるようになることで，作業の現場においても数値シミュレーションが活用できるようになると考えられる．

そこで本研究では，FDM や FEM を用いた数値シミュレーションで扱う疎行列を対象にした疎行列格納方式 Pattern Compression (PatComp) method と Row Block Packing

(RBP) method を提案し，疎行列の格納に必要となるメモリ使用量を削減する．提案する二つの疎行列格納方式は，FDM や FEM で生成される疎行列中に，連続した非ゼロ要素が多く存在すること（連続性）や各行の非ゼロ要素の並びに類似性があること（パターン性）等の特徴があることに着目し，その特徴を利用した手法である．提案疎行列格納方式の評価において，いくつかの FEM の行列と，医療への応用例として心臓シミュレーション (UT-Heart)[8] の行列を用いた．疎行列のメモリ使用量を削減することで，疎行列を用いた処理のメモリアクセス効率の劣化等による GPU 上での演算速度の低下が見られる場合もある．しかし提案疎行列格納方式のメモリ使用量は既存手法に対し，最大 31.3%少ないながらも，Sparse matrix-vector multiplication (SpMV) の演算速度は従来手法と遜色無い結果となった．

本稿は，第 2 章で数値シミュレーションと様々な解析手法を俯瞰する．第 3 章で GPGPU の詳細と GPGPU にて使用される既存疎行列格納方式について述べる．第 4 章にて提案圧縮方法の 1 つである Pattern Compression method (PatComp)，第 5 章にてもう 1 つの提案圧縮方式 Row Block Packing method (RBP 法) を提案しそれぞれ評価を行う．第 6 章にてまとめを行う．

第2章　数値シミュレーションと様々な解析手法

2.1　はじめに

本章にて数値シミュレーションとその解析手法について俯瞰する．まず2.2節にて数値シミュレーションの有用性や活用例について説明を行う．そして2.3節にて，数値シミュレーションの代表的解析手法であるFinite Defferent Method (FDM) とFinite Element Method (FEM) について利点欠点を挙げる．またそれぞれの手法が連立一次方程式の求解に帰着することを説明する．2.4節では，連立一次方程式の求解法である直接法と反復法の種類やそれぞれの特色について説明する．FEMとFDMが連立一次方程式の求解に帰着することから，最適な手法を選択することでシミュレーションの高速化に繋がる．その後2.5節にて大規模な数値シミュレーションで多く使用される非定常反復法について記述する．代表的な非定常反復法であるConjugate Gradient(CG)法，Gradient Stabilized(BiCGSTAB)法，Generalized minimum residual (GMRES)法を挙げ，FDMやFEMで扱う疎行列がアルゴリズム中のどの処理にて使用されるか説明を行う．またこの疎行列が全てのアルゴリズムにて，メモリ使用量の視点からボトルネックとなることを述べる．最後の2.6節にて，非定常反復法の支配的な処理であるSparse matrix-vector multiplication (SpMV)にて述べる．

2.2　数値シミュレーション

数値シミュレーションは宇宙の成長過程の検証，地震による被害の予測，自動車や航空機の風洞テスト等，様々な分野で使用されている[9][10][11]．現実では検証不可能な実験や製品試作段階での低コスト化のために数値シミュレーションが使用されており，科学の発展，生産性の向上等に多く貢献している．

数値シミュレーションを行うには，対象となる現象を膨大な数の方程式を用いてモデル化し，時間の変化ごとに全ての方程式を解く必要がある．シミュレーションに用いるモデルや必要とする精度等により，適したシミュレーションの数値解析手法が異なることから，今までに多くの数値解析手法が提案されてきた．また数値シミュレーションによる解析を行う際には，モデルを構築する膨大な数の方程式を解く必要があり，演算時間や演算資源を多く費やすこととなる．そのため一昔前まで数値シミュレーションは，スーパーコ

ンピュータや大規模クラスタを保有する研究機関や一部の大企業でのみ活用されていた．しかし情報科学の発展に伴う，コンピュータの演算性能の飛躍的な向上や低コスト化により数値シミュレーションは身近な存在となった．その情報科学の発展の中でも，省スペースかつ低コストで高い演算性能を得ることが可能な GPGPU の出現は，小規模な組織での数値シミュレーションの実用化に大きく貢献した．現在では，病院内での検診に GPGPU を使用した数値シミュレーションを用いることで，侵襲的な検査や手術を減らし，患者の負担を減らす動きもある．このように数値シミュレーションは一部の人々だけでなく多くの人が身近に感じる技術となっている．

2.3 数値シミュレーションの解析手法

多く数値シミュレーションでは，物理現象を解析する際に微分方程式を解く必要がある．本節において，代表的な微分方程式の解析手法である Finite Defferent Method (FDM) と Finite Element Method (FEM) の概略について述べる．FDM と FEM は最終的に，連立一次方程式の求解に帰着し，連立一次方程式は疎行列とベクトルとして表現される．よって GPGPU を用いて FDM や FEM を用いたシミュレーションを高速化するには，この疎行列とベクトルを GPU メモリへ格納する必要がある．数値シミュレーションのモデルが巨大化または複雑化することにより，この疎行列のサイズは増大するため，疎行列をいかに小容量の GPU メモリへ格納するかが鍵となる．

2.3.1 Finite Defferent Method (FDM)

Finite Defferent Method (FDM) は，微分方程式をテイラー展開を用いて差分商の式に変換し，近似解を得る手法である．

$$\frac{\partial f}{\partial x} = \lim_{\delta x \to 0} \frac{(f(x+\delta x) - f(x))}{\delta x} \tag{2.1}$$

式 (2.1) に微分の定義を示す．$f(x+\partial x)$ と $f(x)$ の差を微小区間 δx を 0 に近づけることで解を得る．FDM では δx を 0 に非常に近い小さな値で置き換えることで，近似解を得る．δx を小さく設定するほど解の精度が高くなるが，その分計算量は増加する．FDM は正方形や直方体を用いて解析領域を表現することから計算が簡潔になり，FEM に比べ計算量，メモリ使用量が少ないという利点がある．一方で正方形または直方体のみを使用することから，複雑な地形や物体を正確に表現しにくいという欠点がある．

FDM を用いて解析を行う際には，各正方要素間の影響を方程式で表し，それらを合成することにより巨大な連立一次方程式を得る．この巨大な連立一次方程式と解くことで対象領域における物理現象を解析することとなる．また連立一次方程式は疎行列とベクトルで表現される．FDM ではある領域は隣り合う領域からの影響のみを考慮することから，連立一次方程式を表す疎行列中の非ゼロ要素の位置には規則性が存在する．一般的に

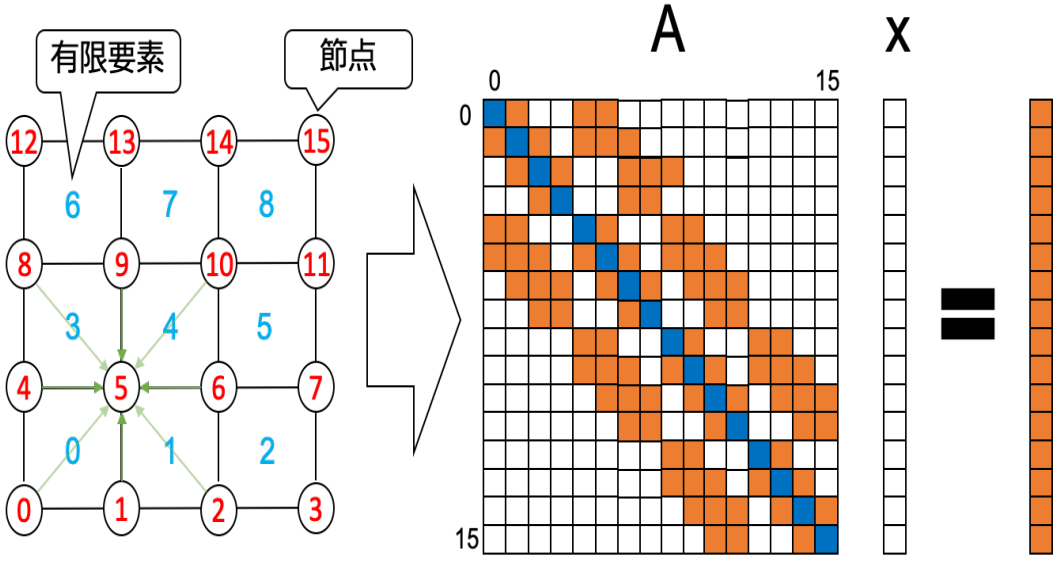

図 2.1: FEM にて扱う疎行列

隣り合う領域間の影響は非ゼロ要素内で連続した非ゼロ要素として表現される．そのため疎行列内には連続した非ゼロ要素が多く存在する．

2.3.2 Finite Element Method (FEM)

FEM は FDM と異なり，様々な要素を用いて対象領域を近似するため，解析の精度が高い．しかし FDM に比べ複雑な要素を用いることから，メモリ使用量，計算量が大きくなる．

FEM では支配方程式を変分原理や重み付き残差法により積分方程式 (弱形式) に変換する．また解析領域を四面体や六面体といった有限要素に分割し，要素の頂点，辺あるいは内部には物理値を持つ節点が配置される．要素内の物理値は各節点の物理値と節点ごとの補間関数により近似される．解析領域全体の積分方程式は，要素ごとの方程式に分割され，要素ごとの積分は上述した要素内の補間関数とガウス積分のような数値積分を用いることで，離散的な要素剛性方程式が構成される．これら要素ごとに作成された方程式を重ね合わせることで，解析領域全体の連立 1 次方程式が得られ，最終的にはこの方程式の求解が必要となる．連立 1 次方程式に現れる行列の各行は節点の物理値に対応し，各列は接続関係にある節点の物理値に対応する．すなわち接続関係にある場合は非ゼロ，その他はゼロとなるため，この行列はその大部分がゼロとなる疎行列である．

FEM も FDM と同様に，隣接する要素からの影響を考慮するため，連立一次方程式を表す疎行列には規則性が存在する．また FEM を用いた解析の方が複雑になりやすく，連立一次方程式のサイズも FDM に比べ大きくなる場合が多い．そのため本研究では FEM

のメモリ使用量削減を目標とする．本稿では主にFEMで生成される疎行列に対するメモリ使用量の削減手法を提案するが，多くの場合FDMにも適用可能である．FDMで生成された疎行列を用いた評価実験は今後の課題とする．

2.4 直接法と反復法

FDMとFEMは連立一次方程式の求解に帰着することから，数値シミュレーション高速化のため，連立一次方程式を解く手法が重要となる．連立一次方程式を解くとは，以下の式において既知の行列Aとベクトルbから未知ベクトルxを求めることを意味する．

$$Ax = b \tag{2.2}$$

行列Aは数値シミュレーションで扱うモデルにより特色が異なるため，様々な連立一次方程式の求解法が存在する．本2.4節では，その中でも代表的な求解手法である直接法と反復法について述べる．

2.4.1 直接法

直接法は行列Aを求める解xが変化しない範囲で変形し，解xを求める方法である．直接法は，コンピュータによる計算誤差を考慮しない場合，有限回の計算で数学的に正しい解が得られることが知られている．行列AがN行N列である場合，直接法の計算量は$O(N^3)$となる．そのため行数Nが分かれば直接法の計算量がどの程度か推定することができるという利点もある．しかしながら現代の数値シミュレーションでは，100万行を優に超える行列を使用することも多々あり，直接法を使用する場合の計算量はおおよそ10^{18}となることから，途方もない演算時間を必要とする．

よく知られる直接法としてGaussの消去法やLU分解法があるが，近年ではLU分解法を選択することが多い．LU分解法は膨大な計算量を必要とするだけでなくメモリ使用量が増加するという問題がある．LU分解法は行列Aを上三角行列Uと三角行列Lに分解する．その後，$Ly = b$を前進代入法で解き，求めたyを用いて$Ux = y$を解くことで解xを求める．しかしながら，FDMやFEMで扱う疎行列をLとUに分解した際に，元々存在した非ゼロ要素の位置以外にも非ゼロ要素が現れる(fill-inと呼ばれる)．そのため行列Aを格納するために必要なメモリ使用量がLU分解前より増加してしまう．またLU分解の計算には依存性が存在し，並列化を行うのが難しいことも知られている．

これらの問題点から，大規模な数値シミュレーションを行う際に直接法を用いることは少なく，反復法により連立一次方程式を解くことが多くなっている．

2.4.2 反復法

反復法は適当な初期値ベクトル x_0 から出発し，何らかの処理を反復して施すことで真の解 x を近似する方法である．反復法は真の解 x と x の近似解との残差 r が閾値 ϵ 以下になるまで反復を繰り返す．この時，残差 r が閾値 ϵ 以下になることを収束したと呼ぶ．

直接法と異なり，反復法で得られる解は近似解である．閾値 ϵ を変化させることで近似解の真の解に対する近似度を決定する．そのため数値シミュレーションに要求される精度に合わせ，閾値 ϵ を決める必要がある．但し閾値を大きくしたとしても，反復法が必ず収束するとは限らない．シミュレーションに複雑なモデルを使用する場合，反復法単体では収束することは稀であり，収束したとしても膨大な反復回数を必要とする．よって用いるモデルに合わせた反復法の選択が重要となり，これまでに様々な反復法が提案されてきた．また反復法の収束性を高めるために，複雑なモデルでは前処理を反復法と併用することが多い．反復法の収束速度は行列 A の固有値分布に依存するため，前処理を用いて固有値分布を改善することで収束性を向上させることが可能である．難しいモデルほど強力な前処理が必要となるが，強力な前処理ほど並列化が困難である．そのため前処理に関する並列化の研究も多く行われている．しかし本研究は連立一次方程式を解く際のメモリ使用量の削減を目的としているため，前処理についての議論は行わない．GPGPU に適した前処理も多く提案されているため，シミュレーションのモデルに合わせて前処理を選択し，提案するメモリ使用量削減手法と併用することが可能である．

一般的に反復法にかかる演算時間，メモリ使用量は直接法に比べ，少ないと言われている．また反復法は並列化が容易であるため，スーパーコンピュータや GPGPU といった並列計算機による高速化の効果が大きい．よって大規模かつ複雑な数値シミュレーションを行う際には反復法を選択することがほとんどである．

反復法は定常反復法と非定常反復法に大別される．以降の 2.4.2.1，2.4.2.2 節にて，それぞれの特色を説明する．

2.4.2.1 定常反復法

様々な定常反復法が提案されているが，有名な手法として Jacobi 法，Gauss-Seidel 法，Successive Over-Relaxation (SOR) 法がある．

定常反復法の特徴として，反復ごとに近似解 x の値のみを変化させ，その他の変数を変化させない．そのためアルゴリズムが容易であり，実装が短時間でできるといった利点がある．

しかし一般的には非定常反復法に比べて収束性が悪い．よって実用的な数値シミュレーションに関しては，演算時間とメモリ使用量の観点から非定常反復法を採用することが多い．そのため非定常反復法を高速化，省メモリ化することが多くの数値シミュレーションの実用化に貢献する．

表 2.1: Feature of each linear solver

評価項目	直接法	反復法	
		定常反復法	非定常反復法
計算量	$O(N^3)$	少ない	やや少ない
メモリ使用量	非常に多い	疎行列内の非ゼロ要素数に依存	
収束性	-	遅い　定常反復法に比べ早い	
対応する解析手法	幅広い	FDM，FEM	

2.4.2.2　非定常反復法

非定常反復法は krylov 部分空間法とも呼ばれ，krylov 部分空間への写像を用いて解 x を求める手法である．krylov 部分空間への写像を用いることで，解の探索範囲に条件が追加され，定常反復法より早く収束することが多い．行列 A の形状により使用される非定常反復法は異なり，形状ごとに適した定常反復法が提案されている．代表的な定常反復法として Conjuate Gradient (CG) 法，Biconjugate Gradient Stabilized(BiCGSTAB), Generalized minimum residual (GMRES) 法が挙げられる．

非定常反復法は並列度の高い処理が多く含まれることから並列計算機を適用しやすく，収束性も良いことから多くの数値シミュレーションで使用されている．実際に GPGPU を用いた非定常反復法の高速化に関する研究が多くなされている [13][14][15].

2.5　様々な非定常反復法のメモリ使用量

FDM や FEM の求解は連立一次方程式を求解に帰着することから，FDM，FEM を用いた数値シミュレーションの高速化は，非定常反復法の高速化と同義である．そのため GPGPU を用いて高速化を行う場合に非定常反復法で用いられるデータを全て小容量な GPU メモリ上へ格納する必要がある．本節では各非定常反復法の特徴を説明したのち，FDM，FEM で生成された疎行列がメモリ使用量の観点からボトルネックとなること説明する．

2.5.1　Conjugate Gradient(CG)

Conjugate Gradient (CG) は非定常反復法の中でも最も代表的な手法であり，行列 A が対称正定値行列の場合に使用される．CG 法のアルゴリズムを Algorithm1 に示す．ここで b は式 (2.2) の右辺ベクトル，x は式 (2.2) の未知ベクトル x，A は式 (2.2) の疎行列を示す．また r は残差を示しており，r が閾値以下となった時点で反復を打ち切り，その時の x の値が解となる．

Algorithm 1 Conjuate Gradient algorithm [12]

1: Compute $r_0 = b - \boxed{\text{A}}x_0$, and $p_0 := r_0$
2: **for** $j = 0, 1, ...,$until convergence **do**
3: $\quad \alpha_j := (r_j, r_j)/(\boxed{\text{A}}p_j, p_j)$
4: $\quad x_{j+1} := x_j + \alpha_j p_j$
5: $\quad r_{j+1} := r_j - \alpha_j \boxed{\text{A}}p_j$
6: $\quad \beta_j := (r_{j+1}, r_{j+1})/(r_j, r_j)$
7: $\quad p_{j+1} := r_{j+1} + \beta_j p_j$
8: **end for**

Algorithm1 中に示す二重四角で囲まれた A に FDM，FEM で生成された行列が使用される．この疎行列 A が CG を解く際のメモリ使用量の観点でボトルネックとなる．Algorithm1 中の α，β はスカラー値であり，r，b，p，x は長さ N のベクトルである．それに比べ疎行列 A は $N \times N$ のメモリ領域を必要とする．全ての変数が倍精度小数点数を使用し，1 つの倍精度小数点数の格納に 8byte 消費すると過程すると，CG のメモリ使用量は $8(2+4N+(N \times N))$ となる．この式から A がメモリ使用量の観点でボトルネックとなることは明確である．

2.5.2 Biconjugate Gradient Stabilized(BiCGSTAB)

Algorithm 2 BiCGSTAB algorithm [12]

1: Compute $r_0 = b - \boxed{\text{A}}x_0$; r_0^* arbitrary;
2: $p_0 := r_0$
3: **for** $j = 0, 1, ...,$until convergence **do**
4: $\quad \alpha_j := (r_j, r_0^*)/(\boxed{\text{A}}p_j, r_0^*)$
5: $\quad s_j := r_j - \alpha_j \boxed{\text{A}}p_j$
6: $\quad \omega_j := (\boxed{\text{A}}s_j, s_j)/(\boxed{\text{A}}s_j, \boxed{\text{A}}s_j)$
7: $\quad x_{j+1} := x_j + \alpha_j p_j + \omega_j s_j$
8: $\quad r_{j+1} := s_j - \omega_j \boxed{\text{A}}s_j$
9: $\quad \beta_j := (r_{j+1}, r_o^*)/(r_j, r_0^*) \times \alpha_j/\omega_j$
10: $\quad p_{j+1} := r_{j+1} + \beta_j(p_j - \omega_j \boxed{\text{A}}p_j$
11: **end for**

Biconjugate Gradient Stabilized (BiCGSTAB) は非対称行列に対する非定常反復法の 1 種である．BiCGSTAB 法のアルゴリズムを Algorithm2 を示す．非対称行列を対象とする手法の中で，計算量が少なく，メモリ使用量も少ないことで知られている．ただし残差は単調に減少せず，残差が発散して収束しない場合もある．

BiCGSTAB は 6 個の長さ N のベクトル b，r，r_0^*，p，s，x，3 個のスカラー α，β，ω と大きさ $N \times N$ の疎行列を使用する．よって倍精度浮動小数点数に 8byte 必要と過程すると，使用するメモリ量は $8(3 + 6N + (N \times N))$ となる．BiCGSTAB においても疎行列格納に必要となるメモリ量が，全体のメモリ使用量の多くを占める．

2.5.3 Generalized minimum residual (GMRES)

Algorithm 3 GMRES algorithm [12]

1: Compute $r_0 = b - \boxed{\text{A}}x_0$, $\beta := |r_0|_2$, and $v_1 := r_0/\beta$
2: Define the $(m+1) \times m$ *matrix* $H_m = \{h_{ij}\}_{1 \le i \le m+1, 1 \le j \le m}$.*Set* $\bar{H}_m = 0$.
3: **for** $j = 1, 2, ..., m$ **do**
4: Compute $\omega_j := \boxed{\text{A}}v_j$
5: **for** $i = 1, ..., j$ **do**
6: $h_{ij} := (\omega_j, v_i)$
7: $\omega_j := \omega_j - h_{ij}v_i$
8: **end for**
9: $h_{j+1,j} = |\omega_j|_2$. If $h_{j+1,j} = 0$ *set* $m := j$ *and go to* 12
10: $v_{j+1} = \omega_j / h_{j+1,j}$
11: **end for**
12: Compute y_m the minimizer of $|\beta e_1 - \bar{H}_m y|_2$ and $x_m = x_0 + [v_1, \cdots, v_m]y_m$

Algorithm3 は非定常反復法の 1 種である Generalized minimum residual (GMRES) のアルゴリズムを示している．GMRES も BiCGSTAB と同じく非対称行列を対象とした非定常反復法である．GMRES は BiCGSTAB と異なり，残差が単調に減少することから発散することがなく，最もロバストな非定常反復法として知られている．そのため収束性が悪い問題に対しては GMRES が多く使用される．しかし残差を単調減少させるため，保持する情報量が増えることから BiCGSTAB に比べメモリ使用量と計算量が増加する．

Algorithm3 では，CG，BiCGSTAB と違い，2 つの行列 A，H_m を使用する．最大の反復回数を m とすると H_m は m 行 m 列の行列となる．また反復ごとにベクトル v を保存することから，v も最終的に N 行 m 列の行列となる．3 つの行列の他に 4 つのベクトル b，r，x, ω と 1 つのスカラー β を使用することから，GMRES の最大メモリ使用量は $8(1 + 4N + (N \times N) + (m \times m) + (N \times m))$ となる．ここで全ての値を倍精度浮動小数点数として格納することを仮定している．最大反復回数 m を大きく設定する H_m や v の領域が大きくなるため，近年ではある程度の反復回数で反復を打ち切り，その時点での解 x を初期値として GMRES を解きなおす Restart 付き GMRES が主流となっている．この場合，最大反復回数 m は多くても 100 回程度に設定されることが多く，H_m，v がメモリ

使用量的に大きなボトルネックとなることは少ない．結局，疎行列 A が支配的なメモリ使用量のボトルネックとなる．

2.6 Sparse matrix-vector multiplication (SpMV)

2.4.2.2 節にて述べた通り，数値シミュレーションを解く際の重要な手法である非定常反復法のメモリ使用量的なボトルネックは FDM や FEM で生成される疎行列の格納である．多くの非定常反復法において二重線の四角で囲まれた疎行列 A は，Algorithm1，2，3 に示すように，ベクトルとの積 Sparse matrix-vector multiplication(SpMV) で使用される．

$$Ax = y \tag{2.3}$$

式 2.3 に Sparse matrix-vector multiplication の定義を示す．SpMV は疎行列 A とベクトル x の積 y を求める演算である．非定常反復法における SpMV は巨大な疎行列とベクトルの積となることが多いため，CPU による逐次処理の場合，非常に長い演算時間が必要となる．SpMV は反復法中の反復毎に実行されるため，非定常反復法の高速化のために重要な処理として知られている．この問題を解決するため，GPGPU を用いた SpMV の高速化が多々行われる [16][17][18][19]．SpMV では疎行列内の非ゼロ要素の情報のみを用いるため，非ゼロ要素の情報のみを効率よく格納することで，疎行列格納に必要となるメモリ使用量を削減できる．疎行列格納に必要なメモリ使用量を削減することが，2.4.2.2 節で述べたように，非定常反復法の大幅なメモリ使用量削減にも繋がる．しかしながら，ただメモリ使用量を削減するだけでなく GPGPU による SpMV の演算性能が低下しないよう考慮して疎行列を圧縮する必要がある．GPGPU における効率的な SpMV の高速化，省メモリ化のため，疎行列格納方法が多く検討されている [20]．

2.7 おわりに

近年，数値シミュレーションが様々な分野で使用され，科学の発展，生産性の向上に貢献している．数値シミュレーションでは解明したい物理現象を表すために，非常に多くの方程式でモデル化し，この方程式を解くことで現象の変化を明らかにする．方程式を解くために様々な手法が提案されているが，総じて膨大な演算時間がかかるため，並列計算機による高速化が行われる．従来，大型並列計算機を用いて数値シミュレーションを解いていたが，GPGPU と呼ばれる技術の出現により，数値シミュレーションを省スペース，低コストで実用化することが可能となった．現在では GPGPU の発展により数値シミュレーションは非常に身近な技術として活用されている．

微分方程式により表現された物理現象を解析する代表的な手法として FDM と FEM が有名である．この 2 つの手法は対象の領域を分割し，領域間の関係性を方程式で表すこと

により解析を行う．その際に分割された領域または節点ごとに立てられた方程式は全て合成され，1 つの連立一次方程式の形で表現される．よって FDM や FEM を用いた数値シミュレーションの求解は連立一次方程式の求解に帰着する．

連立一次方程式を効率よく解くために様々な手法が提案されているが，これらの手法は直接法と反復法に大別される．直接法は有限回の演算で厳密解が得られるという利点があるが，計算量，メモリ使用量が反復法に比べ非常に大きくなるという欠点がある．反復法で求める結果は厳密解ではなく近似解であるが，計算量，メモリ使用量は直接法に比べ非常に少ないという利点がある．そのため大規模なシミュレーションを解く際には，連立一次方程式のサイズが非常に大きくなることから，多くの場合反復法が使用される．反復法には定常反復法と非定常反復法が存在する．非定常反復法は複雑な処理が必要になるが，複雑な問題においても高精度な解が求まることから，実用的なシミュレーションにおいて選択されることが多い．本研究では非定常反復法による連立一次方程式求解を想定している．

FDM，FEM で生成される連立一次方程式は疎行列とベクトルの形で表現され，この疎行列とベクトルを非定常反復法の入力とすることで連立一次方程式を求解する．多くの非定常反復法において，連立一次方程式を表す疎行列を格納するためにメモリ領域の多くを使用する．大規模なシミュレーションにおいては疎行列は非常に大きくなり，行数 100 万を超えることも珍しくない．そのため GPGPU を用いた非定常反復法の高速化の際には，疎行列を圧縮し小容量の GPU メモリへ格納する必要がある．

非定常反復法において疎行列はベクトルとの積 (SpMV) で使用される．SpMV は非定常反復法において支配的な処理と知られ，SpMV の演算性能が重要になる．GPGPU を用いて高速化を行うために，疎行列を圧縮しメモリ使用量を削減する必要があるが，この際に圧縮による SpMV の演算性能を考慮して圧縮する必要がある．

第3章　GPGPUと疎行列の格納方式

3.1　はじめに

本章ではGPGPUと様々なデータ圧縮手法，そしてGPU上でのSpMVで用いられる疎行列格納方式について述べる．3.2節ではGPGPUについて説明し，CPUとGPUの違いについて述べる．またGPUは世代，種類によって演算を行うコア数やメモリバンド幅，レジスタの数などが大きく異なる．アーキテクチャの違いによる演算性能の違いについて3.2節で言及し，その後GPUを用いて演算を行うためのプラットフォームCompute Unified Device Arcgutecture(CUDA)の説明を行う．続いて，実際のアプリケーションにおけるCPU-GPU間のデータ転送を含めた演算の流れについて説明する．GPUを用いて演算を行う際，演算に必要なデータをGPU上のメモリへ事前に格納しておく必要があるが，FEMで生成される疎行列が巨大な場合，GPUメモリに格納しきれず，演算性能の低下を引き起こす可能性がある．そこで疎行列をいかに省メモリに格納するかという問題が存在する．

3.3節では，様々な分野で使用されている代表的なデータ圧縮手法について俯瞰する．そして疎行列に対して代表的なデータ圧縮手法への有効性を検討する．

3.4節においてSpMVで使用される疎行列格納方式についてまとめる．FDMやFEMで得られる行列はゼロ要素を多く含む疎行列となる．SpMVが必要としないゼロ要素を可能な限り削減し，非ゼロ要素のみを格納することがメモリ使用量削減に繋がる．また使用する疎行列格納方式によりGPGPUを用いたSpMVの演算性能は大きく異なる．疎行列格納方式は代表的なデータ圧縮手法とは異なり，非ゼロ要素の情報のみをいかに少ないデータ量で格納するかを重視している．まず最初に疎行列格納方式に対して要求される条件についてまとめを行う．

要求された条件の中で，様々な疎行列格納方式が多く提案されている．一般的にメモリ使用量が多い疎行列格納方式ほどGPU上でのSpMV演算性能は高くなる．これはGPUに適したメモリアクセスを実現するため，SpMVに無駄なパディングを使用するからである．しかしながら近年の数値シミュレーションの大規模化に伴い，高い演算性能を保ったまま，さらにメモリ使用量が少ない疎行列格納方式が望まれている．メモリ使用量をさらに削減することは数値シミュレーションの規模拡大や精度向上に大きく貢献する．これまでに提案された既存疎行列格納方式が演算性能，メモリ使用量にどのような影響を与えるのか，3.4節で説明を行う．

3.2 General-purpose computing on graphics processing units (GPGPU)

3.2.1 GPGPU の概要

General-propose computing on graphics processing unitus (GPGPU) は，従来描画処理を担当していた Graphics Processing Unit (GPU) を汎用的なアプリケーションの高速化に使用する技術である．CPU のクロック周波数が向上しない現状を踏まえ，非常に多くの演算コアを有する GPU を用いたアプリケーションの高速化が様々な分野で試みられている．表 3.1 に各世代ごとに GPU の仕様を示す．最も新しい GPU である Tesla V100 では 5000 個を超える演算コアを有しており，並列度の高いアプリケーションにおいて顕著な高速化を達成することが可能である．単精度浮動小数点数を用いた場合の理論ピーク性能が 15.7TFLOPS であることからも，GPU の演算性能の高さを確認することができる．またスーパーコンピュータ等の大型演算器と比べ，演算性能当たりのコストが導入，維持の面で低いことが知られている．GPU は図 3.1 のようなグラフィックスカードを PCI スロットに差し込むことで容易に使用可能である．

しかしながら単純にアプリケーションを並列化しただけでは，GPU の高い演算性能を引き出すことは困難であり，GPU 上でのメモリアクセスの最適化，キャッシュヒット率の向上等，様々な最適化が必要となる．

図 3.1: Graphic card

表 3.1: GPU specification

Product	Tesla K40	Tesla M40	Tesla P100	Tesla V100	Xeon Platinum 8280
Number of SMs	15	24	56	80	-
Threads per Warp	32	32	32	32	-
Number of core	2,880	3,072	3,584	5,120	56
Clock	875MHz	1,114MHz	1,480MHz	1,530MHz	4,000 MHz
Peak FP32 TFLOPS	5	6.8	10.6	15.7	-
Peak FP64 TFLOPS	1.7	0.21	5.3	7.8	1.6
Memory size	12 GB	24GB	16GB	32GB	最大 1TB
Cache size	1,536KB	3,072KB	4,096KB	6,144KB	38MB
Shared memory size	48KB	96KB	64KB	96KB	-

3.2.2 CPU と GPU の違い

まず CPU と GPU の大きな違いは，搭載している演算コア数である．現在最大の演算コア数を有する CPU は AMD の EPYC 7742 であり，コア数が 64 である．それに比べ GPU では，Tesla V100 で 5120 個の演算コアを搭載しており，CPU との演算コア数に比べ非常に多い．Tesla V100 の最大クロック周波数は 1.53GHz と CPU と比べ 3 分の 1 程度だが，理論的には 5120 並列で動作可能であり，高い高速化率が期待できる．

また GPU と CPU ではメモリ構造が大きく異なる．グラフィックスカードはデバイスメモリと GPU チップから構成される．デバイスメモリと GPU チップ間のメモリバンド幅は一般的な Intel/AMD 等の x86 系のプロセッサが有するものと比べ，6 倍から 7 倍と大きくなっている．GPU で演算を行う際，GPU チップ上の複数の演算装置はデバイスメモリからデータの読み書きを行い，演算を行うこととなる．しかしながら，非常に多くのコアがデバイスメモリへアクセスすることから，このメモリバンド幅は十分とは言えない．現に GPU 内での演算単位 Warp が 2 から 4 サイクルで実行が可能であるのに対し，GPU チップ上の演算装置がデバイスメモリからデータを読み込みを行うために 400 から 600 サイクル要すると言われている．このことから一般的に GPGPU による高速化はメモリアクセス性能に律速される．メモリアクセスの性能を向上させるため，GPU ではデバイスメモリに加えてシェアードメモリ，テクスチャメモリ，コンテキストメモリが GPU チップ上に搭載されている．この 3 つの高速なメモリを効率的に使用することが GPGPU による高速化において非常に重要である．

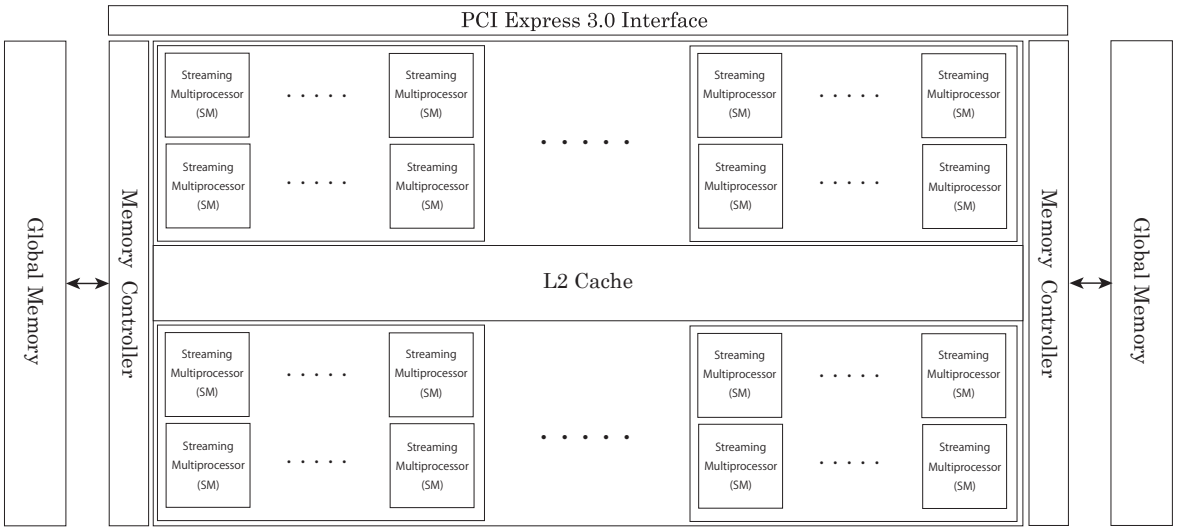

図 3.2: GPU Architecture

3.2.3 GPU アーキテクチャ

図 3.2 に GPU アーキテクチャの略図を示す．GPU の高い演算性能は GPU チップが有する多数の演算装置によってもたらされる．図に示すように GPU チップ上には Streaming Multiprocessor (SM) と呼ばれる演算装置が複数搭載されている．また GPU チップ上の全ての SM で一つの L2 キャッシュを共有する構造である．チップ外に存在する Global Memory からの読み込みを行う際には，一度 L2 キャッシュにデータが格納された後，SM への読み込みが行われる．

図 3.3 に SM 内部の構造を示す．SM 内は複数の CUDA core と呼ばれる GPU 上で最小の演算装置で構成されている．そして各 SM ごとに L1 キャッシュ，Texture Memory が搭載されている．L1 キャッシュは Shared memory と呼ばれる高速なメモリとして使用することも可能である．Shared memory として使用する際にはユーザが明示的にデータの格納を記述する必要がある．また Tesla V100 の場合には SM ごとに 4 つのレジスタと L0 Instruction Cache を備えている．

3.2.4 CUDA

Compute Unifiled Device Architecture (CUDA) は，NVIDIA が提供する GPU 向けの統合開発環境であり，コンパイラやライブラリ等で構成されている．CUDA は CPU のメインメモリから GPU のデバイスメモリにデータを転送のタイミングや展開する Thread と呼ばれる演算の最小単位を明示的に記述することが可能である．

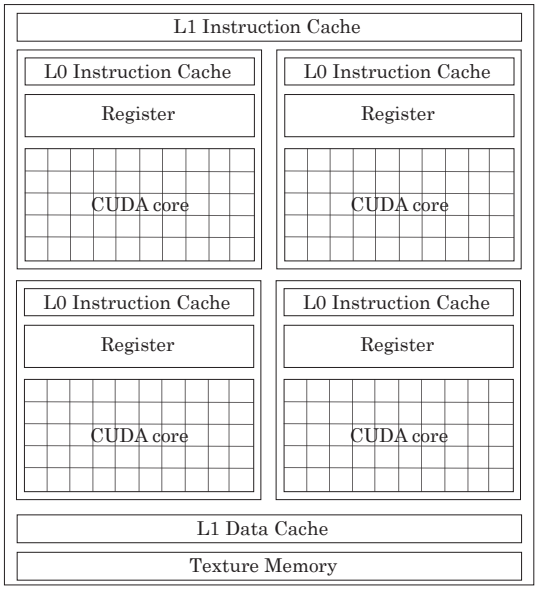

図 3.3: Streaming Multiprocessor

CUDA では複数の Thread を展開し，同時実行することで並列計算を行う．この Thread を管理するため Grid、Block と呼ばれる階層を用いる．図 3.4 のように最上位には Grid 層があり,Grid は複数の Block で構成される．そして Block は演算の最小単位である Thread で構成される．各層の Block 数，Block 内のスレッド数を指定することで，全体 Thread 数を決定することが可能である．

各 Thread は kernel 関数に記述された処理を並列実行する．全ての Thread が同時に kernel 関数を実行するため，各 Thread の区別する必要があり，その区別にはビルドイン関数が使用される．

3.2.5 CUDA のスレッドと Streaming Multiprocessor の対応

GPU チップ上に存在する CUDA コアは最新の GPU でも 5120 コアである．しかし CUDA では Thread を数十から数百万個同時に使用することが可能である．図 3.5 に CUDA の Thread と GPU チップ上の SM との関連性を示す．まず 32 個の Thread を Warp と呼び，SM 上で実行される最小の単位となる．表 3.1 に示すように Tesla GPU では世代にかかわらず Warp 内の Thread 数は 32 個である．また CUDA で宣言した BLOCK は一つの SM で実行される．よって一つの BLOCK を構成する複数の Warp も同 SM 内で実行されることとなる．図 3.5 の GPU チップのように 1 つの SM には複数の BLOCK がキューイングされ，BLOCK を構成する Warp が順次実行されていく．Warp 上の Thread にメモリアクセス等によるストールが生じた際には，実行中の Warp を退避させ，キューから次の Warp

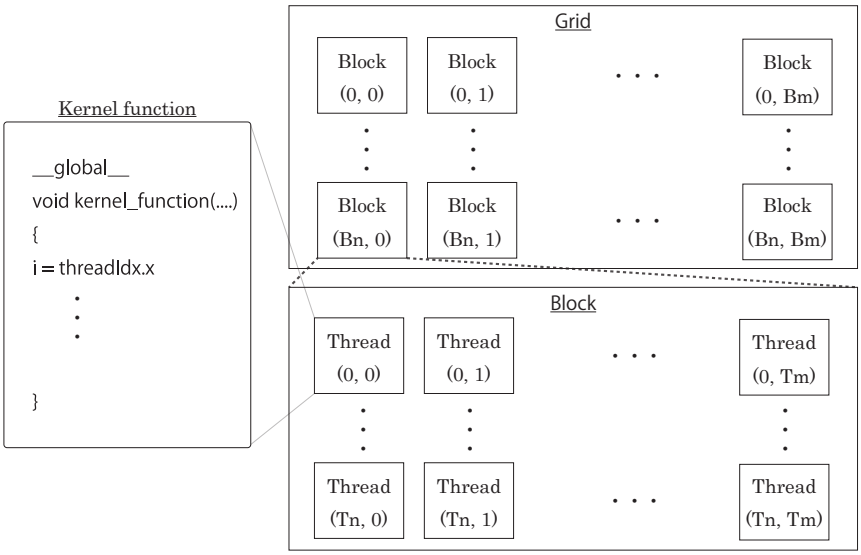

図 3.4: CUDA の階層構造とカーネル関数

を取り出し実行する．GPGPU ではメモリアクセスが高速化のボトルネックとなることから，CUDA では可能な限りメモリアクセスを隠ぺいする Warp scheduling が行われる．このように一つの SM を複数の Warp で共有することで CUDA core より多い Thread を処理することが可能である．

3.2.6 GPGPU を用いた演算の流れ

図 3.6 に GPGPU を用いる際の CPU と GPU 間のデータ転送の流れを示す．図 3.6 の “1. Data transfer CPU - GPU” のように，GPGPU を用いて処理を高速化する際には，処理を行う前に CPU メモリ上のデータを Global Memory へ転送する必要がある．転送後 GPU 上の GPU chip 上の各 CUDA core はカーネル関数を実行する．その際に各 CUDA core は必要なデータを Global Memory から読み出す (図 3.6，“2. Data reading from Global Memory”)．その後，図中の “3. Data writing from GPU Chip” のように，各 CUDA core はカーネル関数の結果を Global Memory へ書き出す．最後に Global Memory 上の Result を CPU Memory へ転送し，GPU による処理は終了となる (図 3.6，“4. Data transfer GPU - CPU”)．

GPU による演算に必要なデータは事前に Global Memory へ格納しておく必要があることから，Global Memory へ格納しきれない程のデータを扱う場合には，対策が必要となる．2 章で述べたように，FEM で生成される疎行列は非常に巨大であり，GPU による演算を行う際に Global Memory へ格納しきれない可能性がある．その際には Global Memory

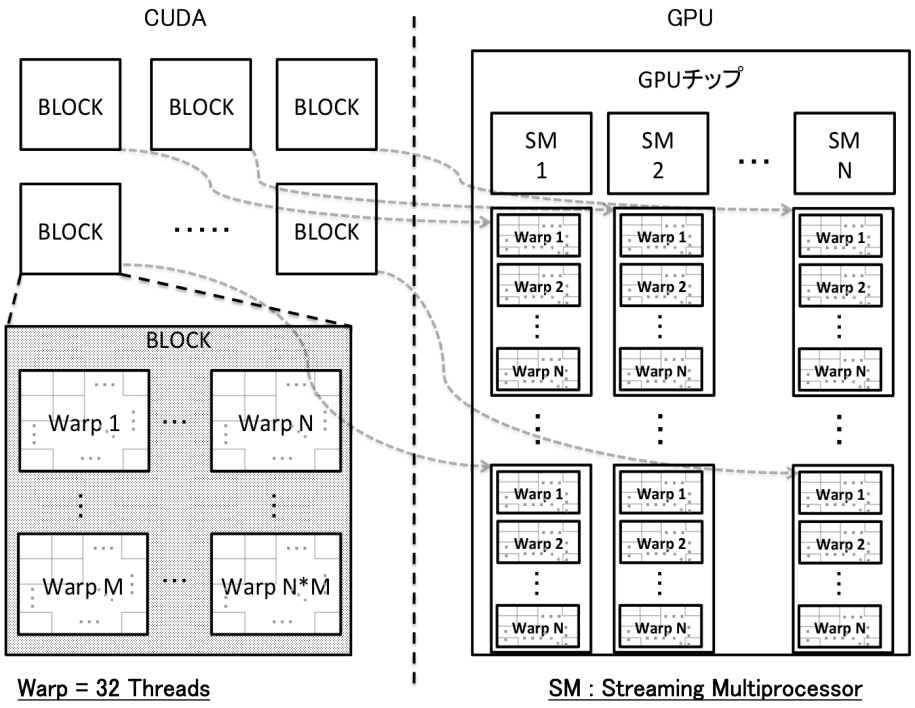

図 3.5: CUDA の Thread と Streaming Multiprocessor の対応

にあるデータを CPU メモリにあるデータと入れ替える必要があり，GPGPU において大きなボトルネックとなるデータ転送が頻発することとなる．そのため GPU による効率的な高速化を行うためには Global Memory に格納可能なサイズのデータに加工し，CPU メモリから Global Memory へ転送する必要がある．FEM で生成される疎行列に対し，どのようにメモリ使用量を削減する方法が適しているのか，次節より検討する．

3.3 様々なデータ圧縮手法

現在，ストレージ容量の節約やネットワークトラフィックの低減等様々な目的のため，コンピュータで使用されるデータ量を削減するデータ圧縮手法が数多く提案されている．本節では，様々なデータの圧縮に使用されている代表的なデータ圧縮手法の疎行列に対する有効性を検討する．疎行列は “0” と実数で構成される数列である．この数列に対してデータ圧縮手法が有効であるか考察する．前提条件として，疎行列は SpMV で使用されることから可逆圧縮である必要がある．

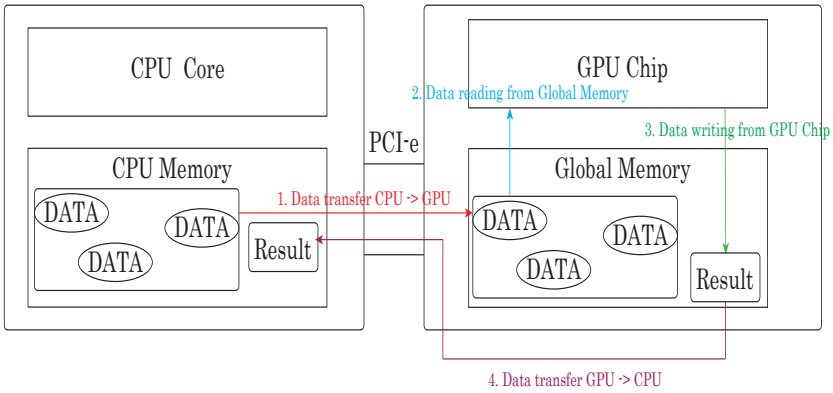

図 3.6: Data flow between CPU and GPU

3.3.1 記号の出現率を用いた圧縮

ハフマン符号化や算術符号化では，記号列中の記号の出現率に基づき，圧縮を行う．記号列中に同じ記号が多く現れる場合，高い圧縮率となる．

3.3.1.1 ハフマン符号化

ハフマン符号化は 1952 年，D. A. Huffman によって提案された手法 [21] である．ハフマン符号化は，圧縮を行う記号列中の出現率の高い記号には短い符号を，出現率の低い記号には長い符号を割り当てる圧縮手法である．出現率の最も低い記号二つに “0”，“1” を割り当て，二つの記号の出現率を合算し，1 つの記号として以降扱う．また同じように出現率の最も低い 2 つの記号を選び’“0”，“1” を割り当てる，最後に出現率を合算する．この処理を記号の数が 1 つになるまで繰り返し，ハフマン木と呼ばれる木構造を作成する．最終的には出現率が高い記号には短い符号が，低い記号には長い符号が割り当てられることになり，全ての符号に同じ長さの符号を割り当てていた場合に比べデータ量が削減される．

疎行列への適用を考えると，“0” が非常に多く存在することから，0 に対する情報量を削減することは期待できる．しかしその他の数値に関しては出現率に偏りがあるとは考えにくく，大きなデータ削減は困難であると考える．また疎行列を使用した SpMV を GPU で並列化することを考えると，圧縮後の符号列中のどこからどこまでが 1 つのデータなのかを瞬時に判別することが難しく（符号列の先頭から復元していく必要がある），高い並列度を達成することは困難である．

3.3.1.2 算術符号化

算術符号化 [22] は，記号列中の記号の出現確率を考慮し，それぞれの記号，記号列を 0 から 1 間の二つの数値の範囲に割り当てていく圧縮手法である．圧縮後の数値が割り当てられたどの記号または記号列の範囲に位置するか判別し，その記号を復元後，割り当てられていた数値を圧縮後の数値から減算する．この処理を繰り返し全ての記号列を復元していく．しかし問題点として事前に記号列の長さがわかっている必要がある点と記号の種類が多くなると丸め誤差等により正しく判別できない場合がある点が存在する．

算術符号化に関してもハフマン符号化と同様に，0 以外の数値の出現率の偏りがないこと，数列の先頭から復元していく必要があることから GPU での SpMV で使用するには不向きである．

3.3.2 連長圧縮

連長圧縮は，記号列中に同じ記号が連続して出現した場合に，その記号，繰り返し回数の二つの値に置き換える圧縮手法である [23]．例えば “abcddddeaf” という記号列があった場合，連続している “dddd” を “d,4” に置き換えることで 4 つの値だったものを 2 つの値で表すことができ，データ量が削減される．上記の例からもわかるように，連長圧縮では同じ記号が連続して出現することが多い場合の圧縮率が高くなる．

連長圧縮では疎行列内に多く存在する 0 に対し高いデータ削減率が期待できる．連長圧縮を使用した場合，各行の先頭から 0 の数分列番号を加算し，次の非ゼロ要素の列番号を計算する．よって連長圧縮でも先頭から復元していく必要があり，高い並列度を得るためには不向きである．GPU 上で疎行列を扱う際には，メモリ使用量を削減しながらも高い並列度で SpMV が実行できる必要がある．

3.3.3 辞書型圧縮手法

LZ77 や LZ78 等は，すでに出現した記号列に対する参照として圧縮を行うため辞書型圧縮手法と呼ばれる．すでに出現した記号列に対しての参照として部分記号列を表すことから，長い記号列を少ないデータ量に置き換えることが可能である．

3.3.3.1 LZ77

LZ77 は 1977 年，J. Ziv と A. Lempel によって提案された圧縮手法 [24] である．LZ77 は，現在使われている多くの圧縮技術の元となる圧縮手法である．ハフマン符号化のように最初に記号の出現率を求めておく必要がなく，容易に適用できることから，多くのソフトウェアに使用されている．LZ77 では，記号列を先頭から参照していき，過去に出現した部分記号列の並びと同様の並びが記号列中に出現した際には，過去の部分記号列への参照の

形で記号列を表す．具体的には記号列を一致した記号列の先頭位置，一致する記号数，次の記号の 3 つの符号で表す．一致する記号列がない場合は（0，0，記号）の形で表す．例えば “aacabcdabcf” という記号列に対して LZ77 を適用すると，“(0,0,a)(1,1,c)(1,3,d)(1,3,f)” のように過去に出現した記号列の先頭位置とそこからの記号数の形で置き換える．よって同じ記号列の並びが多く出現する場合に LZ77 は高い圧縮率となる．LZ77 では，部分記号列を既出の記号列に置き換える際，先頭から全て一致していないか確認すると多大な時間が必要となるため，Sliding window と呼ばれる遡る範囲を決め，その中で一致した記号列が存在しないか確認する方式が一般的である．

LZ77 は同じ記号の並びが多く出現するような記号列に適した圧縮手法であるが，疎行列において複数回出現する要素は 0 のみであり，非ゼロ要素は全て異なる値となることから LZ77 による高い圧縮率は期待できない．またハフマン符号化と同様に LZ77 でも先頭から記号列を復元していく必要があり，中間部分の記号列を瞬時に復元できない．よって SpMV において高い並列度を得ることは困難である．

3.3.3.2 LZSS

LZSS は LZ77 を改良した圧縮方式である [25]．LZ77 では一致した記号列がない場合，(0,0, 記号) の形で格納していたため，無駄なデータ量が追加されていた．LZSS では一致した記号列が存在するかどうかを表す 1bit と追加することで，無駄なデータ量の増加を抑えている．LZSS は LHZ や ZIP といったファイル圧縮に使用されている．

LZSS は LZ77 を元に提案された圧縮手法であり，LZ77 と同様の理由から GPU での SpMV のために利用するには不向きである．

3.3.3.3 deflate

deflate は LZSS とハフマン符号化を組み合わせた圧縮手法である [27]．ZIP や gzip に採用されており，様々なソフトウェアに採用されている代表的な圧縮手法である．deflate では，最初に記号列を LZ77 を用いて，一致する先頭位置と一致する記号数の形に変形する．その後ハフマン符号化を用いて，一致する先頭位置と一致する記号数をそれぞれ出現率ごとに符号化することで高いデータ量削減率を達成している．

GPU 上で SpMV に使用される疎行列を格納するために不向きな LZSS とハフマン符号化の両方を使用する deflate は，LZSS やハフマン符号化と同様の理由から疎行列の格納方法としては不向きである．

3.3.3.4 LZ78

LZ78 は J. Ziv と A. Lempel により 1978 年に提案された [26]．LZ78 は LZ77 と同様に部分記号列を既出の記号列への参照の形で置き換える手法であるが，辞書を用いる点が LZ77 との差異である．LZ77 では一般的に既出の記号列が存在しないか確認するための遡

る範囲が決められている．一方，一般的に LZ78 では出現した記号列を辞書として登録しておくことで，全ての記号列を格納している LZ78 は LZ77 に比べ，圧縮にかかる時間は短いが，復元にかかる時間が長い傾向がある．

LZ78 では既出の記号列を辞書に登録するが，LZ77 と同様に複数回出現する記号は 0 のみであり，高いデータ削減率は達成できないと考えられる．また，復元する際，先頭から復元していき，辞書をアップデートしながら展開する必要があるため記号間の依存関係も存在する．そして多くの種類の数値を辞書に登録する必要があることから，辞書のサイズが非常に巨大となり，データ圧縮率が低くなると考える．よって疎行列の格納には適していない．

3.3.3.5 Lempel–Ziv–Welch(LZW)

LZW は LZ78 を元に提案されており，辞書を用いた圧縮手法である [28]．主に GIF にて使用されている．LZ78 では使用する辞書を圧縮しながら作成していくが，LZW ではあらかじめ圧縮前に作成する．作成する辞書には，圧縮する記号列に現れると予想される記号を全て登録する．そのため多くの種類の記号を使用する場合には辞書のサイズが巨大化する．記号列の圧縮開始後は，LZ78 と同じく出現した記号の並びが辞書に登録されていないか確認し，されていない場合は登録し，辞書をアップデートしていく．LZW は高速な圧縮アルゴリズムであるが，deflate に比べると圧縮率は悪いと言われている．

LZW は LZ78 を元に作成された圧縮手法であるため，LZ78 と同様の理由から疎行列の格納には不向きな圧縮手法である．

3.3.4 パターンを用いたデータ圧縮手法

3.3.4.1 Frequent Pattern Compression (FPC)

Alaa R. Alameldeen らはキャッシュを有効に活用するため，キャッシュ内データに対する圧縮方式 Frequent Pattern Compression (FPC) を提案している [29]．LZ78 等の辞書型圧縮手法は長いデータ列を対象としているのに対し，FPC は短いキャッシュラインに対する圧縮手法であり，圧縮，伸長のオーバヘッドの削減を目的とした手法である．長いデータ列に対しては圧縮率が高くならない．FPC ではデータのタイプによっては格納時に無駄があることに着目し，それぞれのデータのパターンに合わせ，“Prefix” を冒頭に付加することでデータの bit 数を削減する．例えば非常に小さい整数データの場合，格納のためには 4, 6, 16bit のように小さい bit 数しか必要ないが，一般的には 32bit として格納される．この格納方式はメモリ容量的に無駄が大きいため，FPC ではデータの冒頭にデータが何 bit 使用するか見分けるための “Prefix” を付加してキャッシュに格納する．このような方法と採用することで，キャッシュ内のデータはそれぞれ本来必要とされる bit 数で格納することが可能となり，メモリ使用量が削減される．結果として既存のキャッシュ内データに対する 2 つの圧縮方式より FPC のキャッシュ内のデータ量は少なくなっている．ま

た参考のため，キャッシュ内のデータ全てに deflate を採用した gzip を適用した場合との比較を行っており，gzip の圧縮率の方が圧縮率は高くなっている．

3.3.4.2 Frequent Value Locality and Value-Centric Data Cache (FVC)

Youtao Zhang らはプログラム中に使用される値の局所性が高いことに着目し，キャッシュミス率を改善するための Frequent Value Cache (FVC) を提案している [30]．論文中で SPEC ベンチマークを用いて各プログラムの動作を確認した結果，10 個の異なる値が格納されているメモリロケーションへの参照が全体の 50%以上を占めていることが判明した．また 6 個のベンチマークにおいてこの少数の値に起因するキャッシュミスが平均 50%以上であることも明らかになった．そこで頻繁にアクセスされる少数の値が格納されているアドレスのデータのみを格納するダイレクトマップ方式の FVC を実装し，DMC と併用することで 1%から 68%のミス率の削減を達成した．

FVC はキャッシュミス率削減による演算性能の向上が目的であるが，このデータの局所性を用いることでデータ圧縮手法に応用することも可能である．例えば長いデータ列中に頻繁に現れる部分データ列を辞書に登録しておくことでデータ列中の部分データを辞書へのインデックスに置き換えることができる．おおよそ LZ78 と同じ原理となるが，LZ78 と異なりデータ列を全て辞書に格納することはない．そのため圧縮率に関しては LZ78 の方が高くなると考えられる．

3.3.5 文法圧縮手法

文法圧縮手法は，記号列に対して規則を作成し，圧縮した記号列を規則に則り復元していく手法である．

3.3.5.1 Re-Pair

RePaira は文法圧縮手法として広く知られたアルゴリズムである．文法圧縮手法は，記号列中の頻発する部分記号列を規則で置き換え（規則は別のテーブルに格納し，そのインデックスで記号列を置き換える），木構造的に規則への置き換えを行うことで，元の記号列より短い文字列を生成する．しかし文法圧縮では LZ77 圧縮等のときと同様に数値の並びに規則性が希薄であることから高いメモリ使用量の削減は見込めない．また文法圧縮手法では，終端記号（疎行列では，存在する全ての数値）を格納しておく必要があることから，同じ数値が少ない疎行列においては圧縮が効果的に行えないことがわかる．

3.4 疎行列格納方式

FDM や FEM で生成される行列は多くのゼロ要素を含む疎な行列となる．そのため，そのままの状態の行列を用いて SpMV を実行しようとすると $N \times N$ のサイズの疎行列を格納しておく必要がある．近年数値シミュレーションで使用される疎行列の行数は優に 100 万を超えることから GPU メモリのサイズを考えるとそのままの疎行列を扱うことは困難である．そのため何かしらのメモリ使用量が少ない形に変換して，GPU 上に格納する必要がある．

3.4.1 疎行列格納方式の必要条件

前節にて一般的なデータ圧縮手法を俯瞰した．画像，文章，音楽等の同じ記号の並びが多く出現するデータに対しては，これまでに紹介したデータ圧縮手法は非常に有効である．しかしながら，疎行列へ適用することを考えると，同じ記号の並びが現れる記号が 0 だけであり高いデータ削減率が見込めないこと，圧縮後の記号間に依存関係が生じることで並列処理をする際に高い並列度が得られないことを確認した．GPU 上で SpMV を並列して実行することを考えると，各行に対応する情報を瞬時に各スレッドが読み出せるようになっている必要がある．LZ77 や LZ78 では，先の記号列を復元しない限りその後の記号列が復元できない．そのため SpMV で使用する際に，各スレッドが必要な情報を瞬時に読み出すことができず，並列計算が困難となる．

そのため疎行列を少ないメモリ使用量で格納するためには，代表的なデータ圧縮手法を使用するのでなく，疎行列格納方式を用いて格納することが一般的である．疎行列格納方式は，疎行列内の非ゼロ要素のみを効率よく格納し，少ないデータ量で表すことを目的としている．非ゼロ要素のみの情報を格納するためには，非ゼロ要素の位置（行番号，列番号）の情報を付加する必要がある．しかしながら要素間での依存関係が存在しないため，SpMV を高い並列度で実行可能である．

GPU 上での SpMV に使用される疎行列格納方式に求められる条件は，まず高い並列度で SpMV が実行できることである．一つ一つの非ゼロ要素の情報に依存関係をなるべく少なくすることで，GPU の多数のスレッドが必要な情報を同時に読み込むことができ，高い並列度での SpMV を実現することが可能である．次節から説明する CSR や ELL 等の疎行列格納方式では，各非ゼロ要素の情報が他の要素に依存せず求めることができることから GPU による実装で度々使用される．高い並列度を保ちながらも，メモリ使用量をいかに削減するかという点が疎行列圧縮手法の難しさである．

3.4.2 Coodinate (COO) format

COO 形式は疎行列内の非ゼロ要素の値，列番号，行番号を別々の配列に格納することで，疎行列内の非ゼロ要素の情報のみを格納する．図 3.7 に疎行列を COO 形式で格納し

た例を示す．疎行列内の先頭の非ゼロ要素 “a” の値は “a”，列番号は “0”，行番号は “0” であり，それぞれを $Valeus$，$Columns$，Row の先頭に格納する．その後も同様に各行の非ゼロ要素の値，列番号，行番号を順に配列に格納していくことで，図 3.7 のような 3 つの配列が完成する．COO では疎行列内の非ゼロ要素の情報のみを格納していることから，疎行列全体を格納する場合に比べ，メモリ使用量が非常に少なくなる．

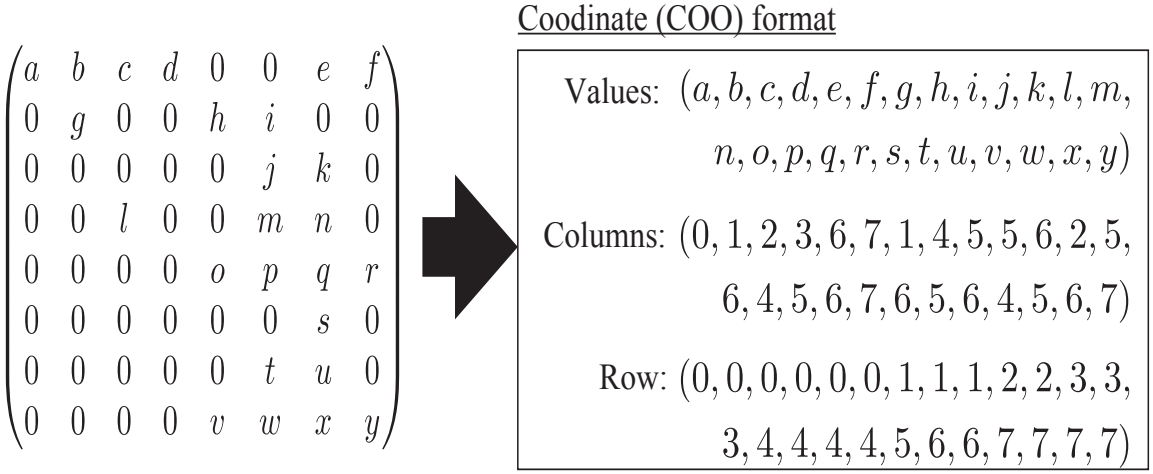

図 3.7: Coodinate (COO) format

COO が提案されて以降，疎行列内の非ゼロ要素の情報のみを効率的に格納する疎行列格納方式が提案されてきた．FDM や FEM で現れる疎行列をファイルに格納する際，シンプルに疎行列を表現できることから，COO 形式で格納されることが多い．一般的には COO 形式で格納されたファイルを読み込み，そこからその他の疎行列格納方式に変換する．

その中でも GPU 上の SpMV に使用することを考慮した場合，Compressed Sparse Row (CSR) と ELLPACK(ELL) が多々採用される．FDM や FEM で生成される疎行列を省メモリで格納することで数値シミュレーションの省メモリ化に繋がるが，格納するデータ構造により GPGPU による SpMV の演算性能は大きく異なる．CSR や ELL が GPU における SpMV で多く使用される理由は，CSR や ELL を用いると高い並列度で並列化でき，SpMV の高速化に繋がるためである．また高い並列度だけでなく，GPU に適したメモリアクセス等の最適化が容易であることから CSR や ELL が使用される．

GPU による SpMV で使用される疎行列格納方式に対する条件は高い並列度で並列実行可能であることである．次節より，GPU において使用頻度が高い Compressed Sparse Row(CSR) と ELLPACK(ELL)[32] の二つ疎行列格納方式について言及する．その後 CSR や ELL の改良版である疎行列格納方式について俯瞰する．

3.4.3 Compressed Sparse Row(CSR)

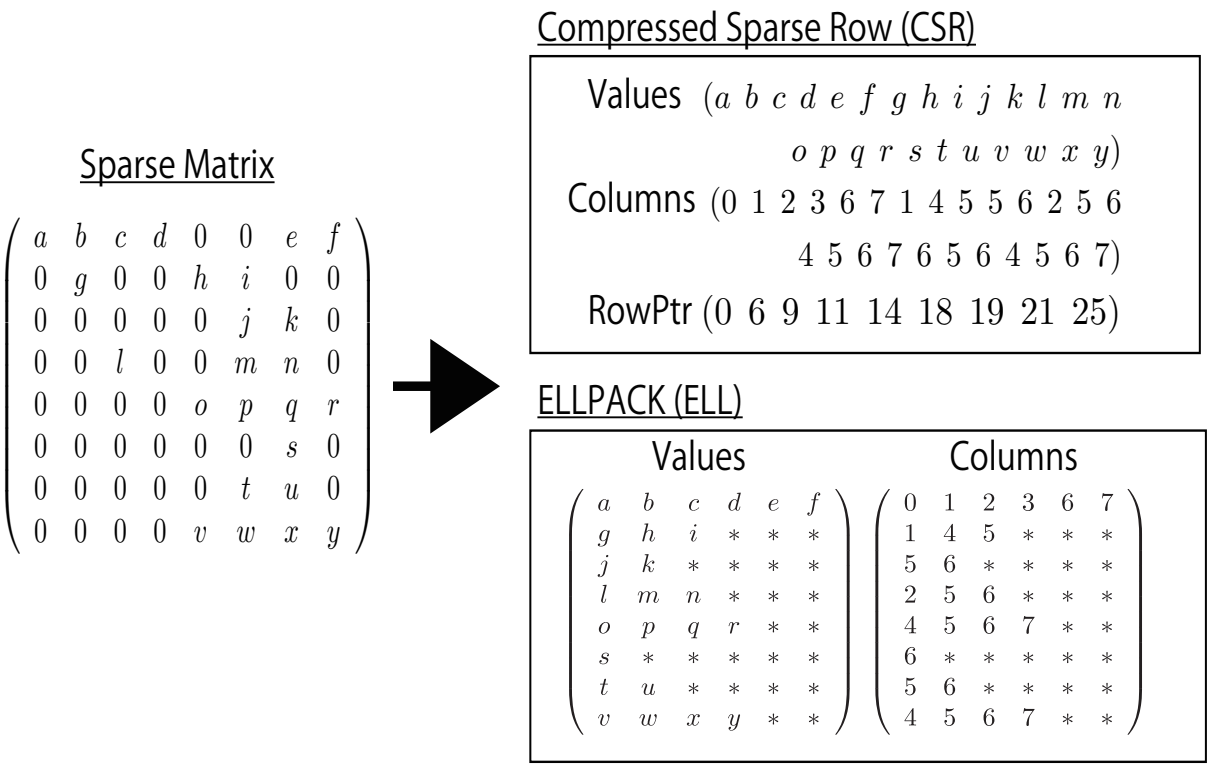

図 3.8: Conventional storage formats for sparse matrices

CSR の長所は，無駄なゼロ要素を一切格納しないため，少ないメモリ使用量で疎行列を格納できる点が挙げられる．また CSR を用いた SpMV の並列処理が容易であることから GPGPU においても度々使用される．

図 3.8，右上に Compressed Sparse Row (CSR) の格納方法を示す．CSR では，3 つの配列を用いて疎行列を表す．一つ目の配列（図中 CSR の四角内，$Values$）は，疎行列内の各非ゼロ要素の値を格納する．二つ目の配列（図中 CSR の四角内，$Columns$）は，疎行列内の各非ゼロ要素の列番号を格納する．三つ目の配列（図中 CSR の四角内，$RowPtr$）は，$Values$ 配列と $Columns$ 配列における，疎行列の各行の最初の要素を示すインデックスを格納する．式 (3.1) に CSR を用いて疎行列を格納した際のメモリ使用量を示す．メモリ使用量の単位は byte である．

$$MemUsage_{CSR} = 8N_{\mathrm{z}} + 4N_{\mathrm{z}} + 4(N+1)\ [byte] \tag{3.1}$$

本稿において，N は疎行列の行数を示し，N_z は疎行列内の非ゼロ要素数を表す．また本稿では，疎行列内の 1 つの値の格納に 8byte(64bit) の倍精度浮動小数点数を用い，列番号等

の 1 つの整数を格納するために 4byte(32bit) の非負整数を使用する．$Values$ と $Columns$ の要素数は疎行列中の非ゼロ要素数 N_z となり，メモリ使用量はそれぞれ $8N_z$，$4N_z$ [byte] で表される．$RowPtr$ の要素数は $N+1$ であり，メモリ使用量は $4(N+1)$ で表される．

Algorithm 4 SpMV code of CSR

1: i = Thread ID　$(i = 0$ to $N)$
2: $dTmpAns = 0$
3: $iRowStart = RowPtr[i]$
4: $iRowEnd = RowPtr[i+1]$
5: **for** $j = iRowStart$ to $iRowEnd$ **do**
6: 　$dTmpAns+ = Values[j] * dVector[Columns[i]]$
7: **end for**
8: $Ans[i] = dTmpAns$

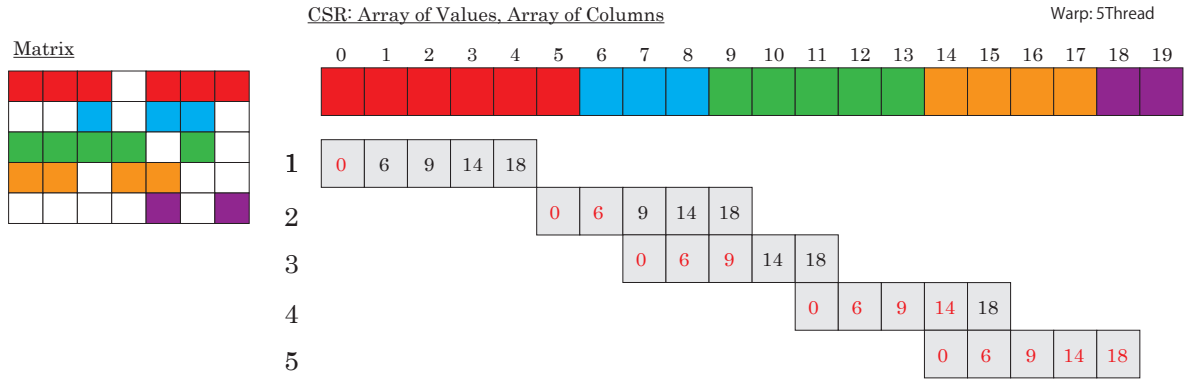

図 3.9: Memory access of CSR to GPU memory

Algorithm4 に CUDA による CSR を用いた SpMV カーネルを示す．Algorithm4 のカーネルを，疎行列の行数分展開された CUDA の Thread で実行することで，並列処理を行う．Algorithm4 中の $Values$，$Columns$ は図 3.8 中の CSR 内，$Values$，$Columns$ に対応する．

まず疎行列の行数 (Algorithm 中，N) 分展開された Thread の ID を i に格納する．これは各 Thread が疎行列の各行に対して SpMV の処理を担当することを意味する．3，4 行目では，疎行列中 i 行目の先頭と末尾の非ゼロ要素のデータが格納されている $Values$ と $Columns$ のインデックスを，$iRowStart$ と $iRowEnd$ に格納する．5 行目の for 文では $iRowStart$ から $iRowEnd$ まで反復することで，i 行目の非ゼロ要素に対する SpMV の処理を行う．6 行目にて $Values$ と $Columns$ から j 番目の値と列番号を読み出し，疎行列とベクトルの計算を行う．最後に $Ans[i]$ に i 行目の SpMV 計算結果を格納し，終了となる．

CSR は非常にメモリ使用量が少ない一方で，GPU 上で SpMV の演算性能が低くなる傾向にある．GPU では同一 Warp 内の Thread は常に同じ処理を同時に実行する．Warp 内の各 Thread の使用するデータが連続したメモリアドレスに格納されている場合，1 回のメモリアクセス処理で読み出すことが可能である．GPU を使用するプラットフォームである CUDA では，このようなメモリアクセス方式をコアレスドアクセス (Coalesced Access) と呼ぶ．しかし，各 Thread が使用するデータが不連続なメモリアドレスに格納されている場合は 1 回で読み出すことができず，最多で Warp 内の Thread 数と同じ回数のメモリアクセスが発生する．このメモリアクセスは演算性能向上の大きなオーバーヘッドとなる．図 3.9 に GPU 上での CSR を用いた SpMV のメモリアクセス例を示す．Matrix 中の色がついている部分が非ゼロ要素を表しており，行ごとに色を変えて表示している．左の行列に CSR を適用すると，$Values$ と $Columns$ は右上のような形で各行の値と列番号を格納する．図 3.9 は各行の先頭の非ゼロ要素とベクトルの値の積を行う際のメモリアクセスを示している．灰色の枠は Thread を表しており，枠内の数字は各 Thread が必要とする，$Values$，$Columns$ に格納されている各行の先頭の非ゼロ要素のデータのインデックスである．説明を単純化するため Warp 内の Thread 数を 5 と仮定する．先頭の Thread が必要とする 0 番目データを読み出す際に，他の Thread は 0 番と連続した 1 番目から 4 番目のインデックスにアクセスする．無駄にアクセスを行ったデータは，その後正しい値を読み出した際に上書きされる．そして次の Thread が必要とする 6 番目のデータを読み出すために，他の Thread は 7 から 9 番目のインデックスにアクセスする．最終的に全ての Thread が必要とするデータの読み込み完了までに 5 回のメモリアクセスを要する．このように CSR では，Warp 内の Thread が必要とするデータを読み出すために複数回のメモリアクセスを必要とすることから，演算性能の低下が起こりやすい．Bell らの論文においては，CSR は ELL に比べ演算性能が低い傾向にあることが報告されている [33].

3.4.4 ELLPACK (ELL)

一方 ELL では，2 つの行列を用いて疎行列を表す．一つ目の行列（図 3.8 中 ELL の四角内，$Values$）では，疎行列内の各非ゼロ要素の値を $Values$ 行列内の対応する行へ格納する．二つ目の行列（図 3.8 中 ELL の四角内，$Columns$）では，疎行列内の各非ゼロ要素の列番号を $Colmuns$ 行列内の対応する行へ格納する．これら二つの行列に左詰めで要素を格納していくため，疎行列内のゼロ要素を削減して格納することが可能である．しかし，二つの行列の列数は，疎行列の行あたりの最大非ゼロ要素数とする必要がある．そのため，疎行列内の各行の非ゼロ要素数のばらつきが大きい場合，計算に使用されない無駄な要素をパディング (図 3.8 中，行列内の $*$) として多く格納する必要があり，メモリ使用量が増加する．一般的に，パディングには “0” を使用するため，本稿においてもパディングには “0” を使用する．CSR では非ゼロ要素のみの情報を保持していたが，ELL ではパディングによって 1 行あたりの要素数を均等にしているため，同一 warp 内の Thread が常に連続したアドレスへアクセス可能である．

Algorithm 5 SpMV code of ELL

1: i = Thread ID　($i = 0$ to N)
2: $dTmpAns = 0$
3: **for**　$j = 0$ to $MaxCol$　**do**
4:　　$dTmpAns+ = Values[i][j] * dVector[Columns[i][j]]$
5: **end for**
6: $Ans[i] = dTmpAns$

Algorithm5 に CUDA による ELL を用いた SpMV カーネルを示す．Algorithm5 のカーネルを，疎行列の行数分展開された CUDA の Thread で実行することで，並列処理を行う．Algorithm5 中の $Values$，$Columns$ は図 3.8 中の ELL 内，$Values$，$Columns$ に対応する．

まず疎行列の行数 (Algorithm 中，N) 分展開された Thread の ID を i に格納する．これは各 Thread が疎行列の各行に対して SpMV の処理を担当することを意味する．3 行目では j を 0 から $MaxCol$ になるまで反復処理を行う．ここで $MaxCol$ は ELL の $Values$ 行列と $Columns$ 行列の列数である．4 行目において疎行列の i 行目の非ゼロ要素を $Values$ から 1 つずつ読み出し，非ゼロ要素の存在した位置を $Columns$ から読み出す．そして疎行列の値とベクトルとの積を求める．最後に i 行目の SpMV の演算結果を $Ans[i]$ に格納し終了となる．

図 3.10 に GPU 上での ELL を用いた SpMV のメモリアクセス例を示す．図中の Matrix を ELL で格納した際の $Values$ と $Columns$ を図中の右側に示している．図 3.9 と同様に色がついている枠が非ゼロ要素を表しており，各行の非ゼロ要素を色を変えて表現している．ELL の $Value$ と $Columns$ を列優先データ順で格納した場合，Warp 内の各 Thread が必要とするデータは連続したアドレスに格納される．そのため各行の先頭の非ゼロ要素のデータを読み出す際も，各 Thread が必要とするデータを 0 から 4 番目の連続したインデックスに 1 回アクセスすることで読み出すことが可能である．パディングされた無駄な要素はメモリ使用量の増加を起こすが，メモリアクセスの観点から見れば非常に効果的である [33].

ELL のメモリアクセスは GPGPU に非常に適しているが，パディングに対する演算は SpMV の結果に影響せず，演算性能を下げる要因となる．そこで Vázquez らは，ELL のパディングされた要素に対する無駄な演算を減らし，演算性能をさらに向上させる手法として，ELLPACK-R (ELL-R) を提案している [34]．ELL-R では ELL に加え，疎行列内の各行の非ゼロ要素数を保持することで，パディングに対する無駄な演算を減らすことが可能である．式 (3.2)，(3.3) はそれぞれ，ELL と ELL-R を用いて疎行列を格納した際に必要となるメモリ使用量である．

$$MemUsage_{ELL} = 8NK + 4NK\ [byte] \tag{3.2}$$

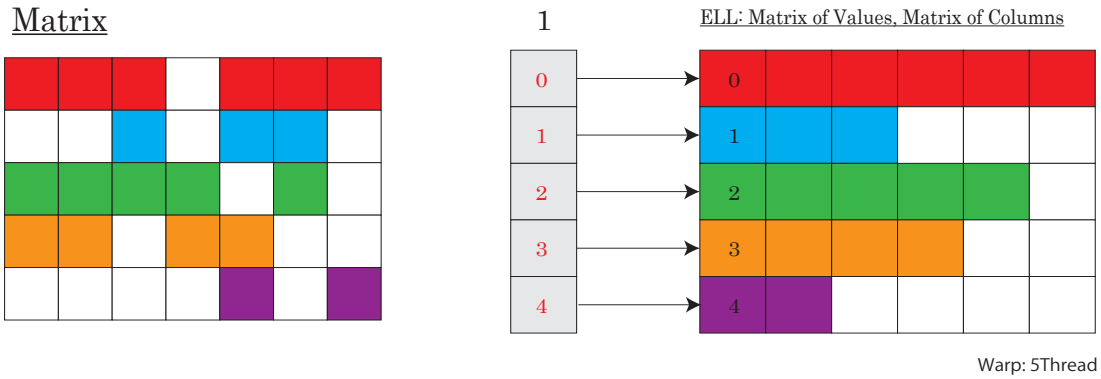

図 3.10: Memory access of ELL to GPU memory

$$MemUsage_{ELL-R} = 8NK + 4NK + 4N\ [byte] \tag{3.3}$$

ここで K は疎行列中，1 行あたりの最大非ゼロ要素数を示す．ELL，ELL-R ともに，$Values$ 行列と $Columns$ 行列内の要素数は NK で表される．そのため，式 (3.2) では $8NK$ が $Values$ 行列のメモリ使用量を，$4NK$ が $Columns$ 行列のメモリ使用量をそれぞれ表している．また，ELL-R では ELL に加え，各行の非ゼロ要素数も保持するため，式 (3.3) では第 3 項として $4N$ が追加される．

3.4.5 Sliced ELLPACK (SELL)

また近年，これまでに述べた疎行列格納方式の高い演算性能を保ったまま，メモリ使用量の削減を試みる研究が行われている．Alexander らは，ELL の無駄なパディングを減らし，メモリ使用量を削減するため，Sliced ELLPACK(SELL) を提案している [35]．SELL では，任意の行数で疎行列を分割し，各分割した部分疎行列に対して ELL を適用する．これにより，各行の非ゼロ要素数のばらつきが大きい場合でも，パディングの量を抑え，メモリ使用量を削減できる．また SELL の派生系として SELL-C-σ [38] があるが，この手法は各行の非ゼロ要素数でソートすることでさらにゼロパディングの数を削減する．しかしながら，SELL は ELL のパディング数を減らすことを目的とした疎行列格納方式であり，FDM や FEM で生成される疎行列の規則性を考慮した情報量の削減等を行うことはない．本研究は疎行列の規則性，パターン性を考慮し，不必要な情報量を削減することでメモリ使用量を削減する手法を提案する．そのため SELL と本稿で提案する手法の方向性が異なるため，比較対象として SELL 実験で用いることはしない．また SELL に対して本研究にて提案する手法を併用することも可能であると考えている．

3.5 位置情報を表す要素へのデータ圧縮手法の適用

FEM で生成されたそのままの疎行列に対して，これまでに提案されてきたデータ圧縮手法の適用は困難であることを 3.3 節にて説明した．しかしこれまでに紹介した CSR や ELL 等の非ゼロ要素の値と位置情報（列番号）を分離させた状態であれば，位置情報に対してデータ圧縮を行うことが可能となる．列番号を格納している配列内の要素には規則性がある場合が多いため，データ圧縮手法が有効である．しかしながら，LZ77 等の圧縮手法をそのまま適用することは困難である．GPU 上での SpMV で使用することを前提としていることから，列番号を圧縮した場合でも高い並列度で SpMV を実行できることが条件となる．LZ77 等の圧縮手法を使用する場合を考えてみると，列番号の先頭から参照していき，過去に出現した列番号の並びをその列番号へのポインタとして格納する．この方法を用いると先頭から順に復元していく必要がある．GPU での SpMV では各行の先頭の列番号を瞬時に復元することができなければ，高い並列度を達成することはできない．そのため，LZ77 のような順に文字列を復元するようなデータ圧縮手法ではなく，複数のポイントからデータを復元できるようなデータ圧縮が必要となる．また文法圧縮を用いた場合でも同様の問題が生じる．文法圧縮では全ての圧縮した文字列を展開してからでないと，GPU 上の各 Thread が SpMV で必要な列番号を得ることができない．そして GPU メモリ上で展開するとなると元々の列番号をそのまま格納した時と同じメモリ使用量が必要となり，圧縮のメリットがない．そのため文法圧縮は疎行列の列番号に対する圧縮には適していない．

3.5.1 Blocked CSR and Blocked ELL

J. Choi らは，Blocked CSR と Blocked ELL を提案している [36]．BCSR と BELL は疎行列内の非ゼロ要素を少ない数の列番号で表すため，疎行列を小さな密行列に分割し，格納する格納方式である．規則に則り，疎行列を小さな密行列に分割することで，各密行列に対し 1 つの列番号から，密行列内の非ゼロ要素の本来の列番号を算出することが可能である．Blocked CSR と Blocked ELL の問題点として，各密行列の行数と列数が固定であること，密行列にする際に無駄なゼロ要素を格納する必要があることの 2 つがあげられる．Blocked CSR，Blocked ELL 共に，第一に演算性能の向上を目指した手法である．そのためメモリ使用量削減を第一に考えた場合，さらにメモリ使用量を削減可能である．本研究は，演算性能を保ったまま既存手法のメモリ使用量を削減することが目的である．

3.5.2 BCSR

Ali らは，CSR 方式に対する疎行列内の連続した非ゼロ要素の列番号を圧縮し，メモリ使用量を削減する手法，Blocked Compressed Row Storage(BCRS) を提案した [37]．図 3.11 に BCRS を用いた格納方法の例を示す．ここで，A_f は行列の値を格納する配列，$Colind$

$$Matrix = \begin{pmatrix} a & b & c & 0 & 0 \\ 0 & d & 0 & e & f \\ 0 & g & h & 0 & 0 \\ 0 & 0 & i & j & 0 \\ 0 & k & 0 & 0 & l \end{pmatrix} \quad \begin{aligned} A_f &= (a,b,c,d,e,f,g,h,i,j,k,l) \\ Colind &= (0,1,3,1,2,1,4) \\ Rowptr &= (0,1,3,4,5,7) \\ Nzptr &= (0,3,4,6,8,10,11,12) \end{aligned}$$

図 3.11: Blocked Compressed Row Storage (BCRS)

は連続した非ゼロ要素の先頭または不連続の非ゼロ要素の列番号を格納する配列である．そして，$RowPtr$ は $Colind$ 内のどの要素が各行の先頭の非ゼロ要素の列番号かを格納し，$Nzptr$ は A_f 配列内のどの要素が連続した非ゼロ要素の先頭，または単体の非ゼロ要素なのかを記憶している．実験結果では，CSR 方式のメモリ使用量とメモリアクセス回数の削減を達成し，連続した非ゼロ要素の列番号圧縮の有効性を示している．しかし，Ali らの研究では，GPU を始めとする並列計算環境におけるの SpMV ではなく，逐次的な SpMV を対象としている．論文中においても，並列計算を行う実装の定義はされていない．そのため，BCRS をそのまま GPU で並列化することは困難であると考えられる．また，多少の変更で BCRS 方式を GPU で実行しても高い性能が出るとは考えにくい．以下に理由を述べる．例えば，単純に GPU 上の 1 Thread が図 3.11 の行列の 1 行を担当し，並列化を試みるとする．各 GPU 上の Thread はまず，$Rowptr$ の値を読み込むことが想定される．次に，$Colind$ から自身が担当するブロックの先頭または単体の要素の列番号を読み込む．その後，$Nzptr$ の値を用いて A_f から値を読み込む際，A_f 内のどの要素からどの要素までが同じ行にあるか並列計算では判別できないという問題が生じる．例えば，図 3.11 の行列の 5 行目の k を計算する場合，k の列番号はわかるが，Thread は並列に動いているために k に対応する $Nzptr$ の数値がわからず，$Nzptr$ のどの部分を読み込めばいいか判別ができない．したがって，このままでは，GPU 上での BCRS を用いた SpMV は実現不可能であると考えられる．

また，著者らもこれまでの研究 [43] において，CSR と ELL 方式を用いて，連続した非ゼロ要素の列番号に対する圧縮法の有効性を検証しており，メモリ使用量の削減に効果的であることが判明している．しかしながら，連続した列番号の圧縮により，Global Memory 上の列番号を保持する配列に対するアクセスが不連続化するため，ELL を用いた SpMV の演算性能が大幅に低下することも明らかになった [43].

3.5.3 Compressed Adaptive ELL (CoAdELL)

M. Maggioni らにより提案された CoAdELL は，AdELL[40] と呼ばれる Warp 内のロードバランスを改善し，ELL の演算性能を改良した疎行列格納方式のメモリ使用量削減を

行っている．SELL の同様に複数行ごとに ELL を適用し，その後各 ELL 内を行中の非ゼロ要素数でソートする．その後，特定の列数で折り返すことで，さらにゼロパディングの数を削減する．最後に列番号を表す行列内の各行に対してデルタエンコーディングを適用することでメモリ使用量を削減している．CoAdELL は変換のためにソート等の処理を多く要するため，ELL 等への変換に比べ，大幅に変換時間が長くなる．

3.5.4 Adaptive Multi-level Blocking (AMB)

AMB は Nagasaka らにより 2016 年に提案された疎行列格納方式である [42]．AMB ではキャッシュサイズを考慮し決定した列番号で疎行列を横方向に分割することで，入力ベクトルに対するメモリアクセス効率の向上とメモリへのアクセスを削減している．その後，複数行ごとに分割し，それぞれを SELL-C-σ [38] を拡張した方式で格納する．最後に列番号が格納されている行列に対して，圧縮を行うことでメモリ使用量が削減される．AMB では演算性能を向上させるため，変換の際にパラメータのチューニングを行っており，複数回の SpMV 実行を必要とする．そのため CoAdELL と同様に長い変換時間が必要となる．

3.5.5 BCCOO

S.Yan らにより 2014 年に提案された疎行列格納方式．疎行列を格納した際のメモリ使用量最も少ない疎行列格納方式の 1 つとして知られている．疎行列を任意のブロックで分割し，ブロックの左上の非ゼロ要素の位置のみを格納することで列番号を示す要素の数を削減している．しかしブロック内にゼロ要素が存在する場合には，値に 0 を格納するため，メモリ使用量が増加する．そのため最適なブロックのサイズを決定するためのチューニングに多くの時間を費やす．またメモリ使用量の削減だけでなく演算性能向上のためのチューニングにも多くの時間が必要となる．[42] の文献においても BCCOO の変換時間は非常に長いと指摘している．

3.5.6 浮動小数点数に対する圧縮

疎行列格納方式は，疎行列内の非ゼロ要素の位置情報をいかに減らすかを考えるため多くの疎行列格納方式で値と位置情報を別々に格納している．本研究では非ゼロ要素の位置情報を削減し，メモリ使用量を削減するアプローチである．一方で疎行列の値を表す浮動小数点数に対して，単精度浮動小数点数の使用等の適用によるメモリ使用量の削減方法も考えられる．この値に対するメモリ使用量削減と本研究のメモリ使用量削減の対象は別のデータであることから，併用して使用することが可能である．

3.6 おわりに

GPGPU は GPU の多くの演算コアによる高い演算性能を活かし，GPU を汎用的なアプリケーションに適用する技術である．GPU は CPU に比べ非常に多くのコアを有しており，非常に高い並列度でアプリケーションを並列実行することが可能である．しかし GPU の Global メモリへのアクセスは低速であることから，高速化のためには GPU Chip 上に搭載される高速なメモリ，Shared memory, Texture memory, Context memory を効率的に使用することが重要である．

NVIDIA GPU を用いる際には CUDA と呼ばれる統合開発環境を使用する．CUDA は記述性の高く，様々なライブラリが用意されているため，簡単に GPGPU を活用することが可能である．CUDA では Grid，Block，Thread の 3 つの階層を用いて並列処理を管理する．Thread は GPU 上の最小演算単位であり，全ての Thread が kernel 関数に記述された処理を並列実行する．

GPGPU では演算を開始する前に，必要なデータを CPU memory から Global Memory へ転送する必要がある．その際，必要なデータが Global Memory へ格納しきれない場合には，CPU-GPU 間のデータ転送が頻発し，データ転送時間が増加する．FEM で生成された疎行列を用いて反復法を解く際には，非常に巨大な疎行列が GPU メモリ上に乗り切らない懸念がある．そのため疎行列に対するメモリ使用量の削減手法が必要となる．

まずデータ量を削減するためには様々なデータ圧縮方式が考えられる．3.3 節ではハフマン符号化や LZ77，LZ78 等の代表的なデータ圧縮手法を俯瞰した．LZ77 等の圧縮手法では復元する際に先頭から順次復元していく必要がある．しかし SpMV による並列計算を GPU で行うこと場合には各スレッドが必要なデータを瞬時に読み出す必要があることから，LZ77 等の圧縮手法を用いて並列処理を行うことは困難である．疎行列に対して適用可能な圧縮方式の条件は，SpMV の際に各スレッドが必要なデータを瞬時に読み出し可能でありながら，メモリ使用量が少ないということである．

そのため，疎行列内の非ゼロ要素の情報のみを効率よく格納する手法が多く提案されている．非ゼロ要素の情報のみを格納するため，値の他に非ゼロ要素の位置情報も格納しておく必要がある．これまでに提案された疎行列格納方式は CSR 系と ELL 系に大別される．CSR 系の疎行列格納方式は ELL に比べメモリ使用量が少ないが，SpMV 演算性能が ELL より低い．一方 ELL 系の疎行列格納方式は SpMV 演算性能が高い代わりに，メモリ使用量が CSR に比べて多くなる．それぞれの欠点を改善するため，BCSR，Blocked CSR, Sliced ELL, Blocked ELL 等の疎行列格納方式が提案されている．また疎行列格納方式内の非ゼロ要素の位置情報を削減するため AdCoELL，AMB，BCCOO 等の疎行列格納方式が提案されている．しかしながら，これらの疎行列格納方式では非ゼロ要素を小さいブロックに分割したのち圧縮を行うため，大幅な圧縮が困難な場合がある．また変換時間が多く必要となるという欠点もある．本研究では，AdCoELL や BCCOO 等の疎行列より広い範囲の圧縮を行うことでメモリ使用量をさらに削減する手法 Patttern Compression（PatComp）法と，変換時間が短いながらも非ゼロ要素の列番号を削減し，メモリ使用量を減らすことが可能な手法 Row Block Packing（RBP）法を提案する．

第4章　数値シミュレーションの時間発展に伴いFEMモデルの形状が変化しない問題に対する疎行列格納方式：Pattern Compression method (PatComp法)

4.1　はじめに

Pattern Compression (PatComp) では，既存の疎行列格納方式で考慮されていない1行中に存在する非ゼロ要素の並びに着目し，メモリ使用量を削減する．2節で述べたように，FEM における疎行列中の非ゼロ要素の出現する位置は，節点間の接続関係と，その節点番号により決定される．FEM モデル中の節点の節点番号が疎行列の行番号に対応し，その節点に隣接する節点の節点番号と同じ列に非ゼロ要素が出現する．

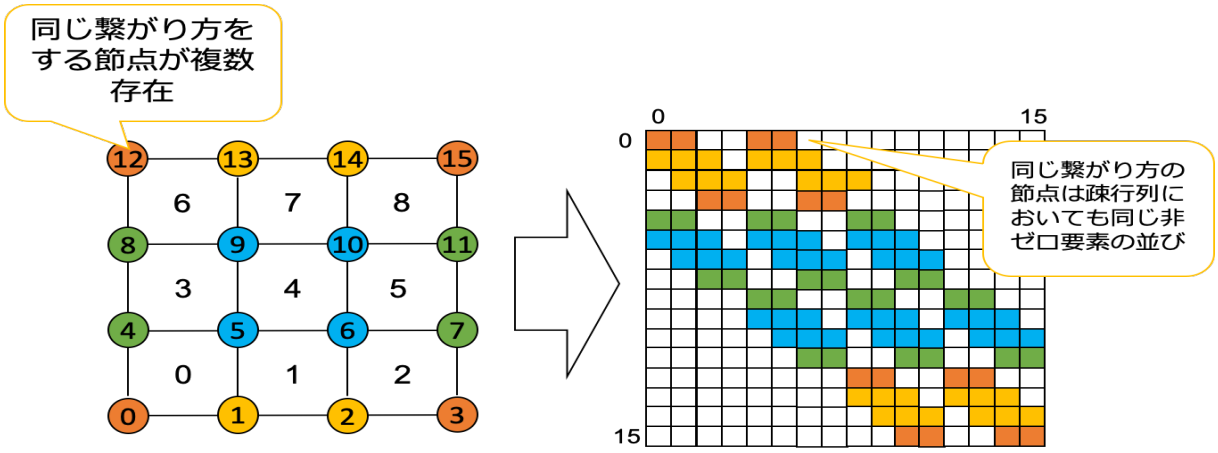

図 4.1: Pattern on matrix by using FEM

一般的に，図4.1 に示すように，複数の節点が，隣接する節点と同じ形で接続する場合，

それぞれの節点に対応する疎行列中の行に，同じ並びで非ゼロ要素が行中に出現する．例えば青色の節点（5，6，9，10）は 8 個の節点に囲まれる形で他の節点と接続している．この場合，疎行列中の青色で示すような同じ非ゼロ要素の並び（先頭の位置のみ異なる）が，各節点の節点番号と同じ行（図 4.1 中の疎行列，5，6，9，10 行）に出現する．よって FEM モデル中の多くの節点が同じ形で他の節点と接続していれば，疎行列中に同じ並びの非ゼロ要素が多く存在することとなり，その非ゼロ要素の並び（パターンと呼ぶ）を 1 つ格納しておくことで，多くの行の非ゼロ要素の位置を表すことが可能となる．

本章では FEM で生成される疎行列では，1 つのパターンを格納しておくだけで，複数の行の非ゼロ要素の位置関係を表現できる利点を用いた PatComp 法を提案する．行毎にパターン化することで，SpMV においても行単位の並列処理を可能としており，GPU における SpMV では従来の疎行列格納方式と同程度の演算性能が期待できる．また SpMV を実行する際に，多くの Thread が同じパターンに複数回アクセスすることになるため，Cache hit rate の向上も期待できる．

既存手法である BCCOO や AMB では，疎行列を小さな行列に分解した部分行列に対して列番号の削減を行っている．しかし小さな範囲内での非ゼロ要素の連続性を用いているため，本来はさらに長く連続した非ゼロ要素の場合削減率は低くなる．本章で提案する PatComp では行中の連続した非ゼロ要素を分割せず，そのまま圧縮することに加え，行ごとのパターンを考慮することで既存手法より広い範囲への圧縮が可能である．そのため既存手法より高いメモリ使用量の削減率が期待される．

しかし広い範囲でのパターンを考慮するため，そのパターンがすでに出現したパターンでテーブルに登録されているか探索するため多くの時間が必要になるため，PatComp への変換時間は長くなることが予想される．そのため PatComp 法はシミュレーションの時間発展ごとに FEM モデルの形状変化がなく，格子の形状が変化せず，疎行列の形も不変な問題に適している．また反復法の収束までに多くの時間が必要となる複雑な問題に対しても有効である．上記の例において，疎行列の変換は数値シミュレーション中で 1 回のみ必要であるため，変換時間に多くの時間が必要となったとしても，演算時間の割合が多くなり，変換時間は支配的ではなくなる．

4.2 節にて FEM で生成される疎行列のパターン性について説明し，4.4 節にて非ゼロ要素のパターン性を活用したメモリ使用量削減手法 PatComp の提案を行う．最後に 4.5 節にて PatComp を用いた SpMV カーネルについて述べる．

その後 4.7 節では，PatComp の目的である既存手法に比べメモリ使用量が大幅に削減できているか確認する．PatComp では既存手法である CSR と比べメモリ使用量を大幅に削減しながらも，同等以上の SpMV 演算性能であることが求められる．

4.7 節の実験では，実際のシミュレーションで用いられた疎行列を用いて，各疎行列格納方式のメモリ使用量，SpMV 演算時間を測定する．また最後に GMRES を使用した評価を行う．GMRES を用いた評価では，PatComp の変換時間を含めた全体の処理時間を測定する．PatComp 法は時間発展に伴い疎行列の形が変化しない数値シミュレーションを対象としているため，疎行列の変換は数値シミュレーションの最初に 1 回のみ実行され

る．そこで PatComp の変換がどの程度の時間を費やすか評価を行う．

4.2　疎行列内の非ゼロ要素のパターン性

4.2.1　非ゼロ要素のパターン性について

FEM で生成される疎行列は，各有限要素に関する解くべき方程式（要素剛性方程式）を重ね合わせた形で構成される．要素剛性方程式を表す疎行列内の非ゼロ要素の位置は，隣接する要素との接続関係により決定される．もし 2 つの要素が同じ形で隣接要素とつながっている場合には，疎行列内の非ゼロ要素の並び（パターン）は一致する．ここでパターンとは 1 行中の非ゼロ要素の位置を，非ゼロ要素の連続数と次の非ゼロ要素までの 0 の数を用いて表した数列である．またパターンが一致するとは，図 4.2 のように，非ゼロ要素の連続数 (図 4.2 中の赤枠)，次の非ゼロ要素までのストライド図 4.2 中の青枠) の並びが一致することを示す．図 4.2 の例では，0 行目のパターンは (2,2,2) であり，1 行目のパターンも (2,2,2) であることから，0 行目と 1 行目は一致している．同様に，2 行目，4 行目も同じパターンであることから，この例において 0，1，2，4 行目のパターンが一致していると言える．構造格子を使用した FEM では要素が規則正しく並べられることから，多くの要素が同じ形で他の要素と隣接すると考えられる．よって，疎行列内の非ゼロ要素のパターンが一致する行が多く存在することとなる．

パターンが一致している多くの行に存在している非ゼロ要素の位置は，各行の先頭の非ゼロ要素の列番号を記録しておくことで，1 つのパターンから復元することが可能である．また疎行列内の全てのパターンを格納しておけば，全ての行の非ゼロ要素の位置関係を復元することができる．そのため疎行列中に存在するパターン数が少ないほど，非ゼロ要素の位置関係を表すために格納しておくデータ量が少なくなり，疎行列格納に必要となるメモリ使用量も削減される．

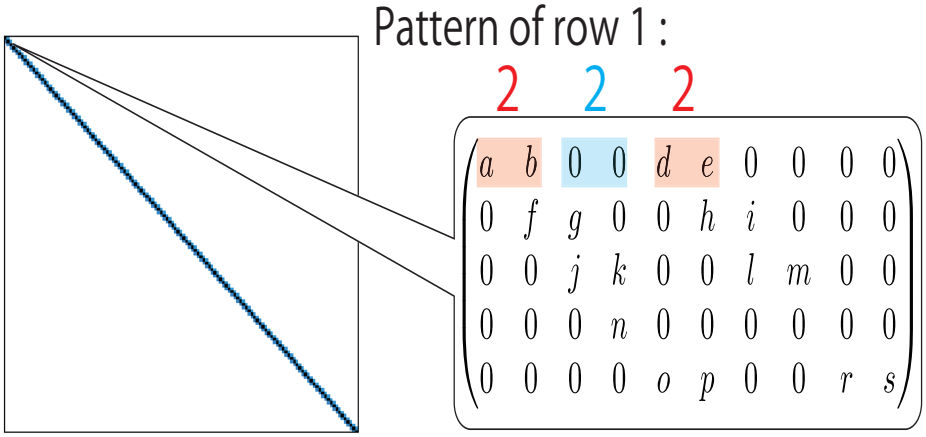

図 4.2: Example: Pattern of a row in a sparse matrix

4.2.2 疎行列内のパターン性に関する予備実験

実際に FEM で生成された疎行列内のパターンが一致する行が多く存在することを確認するため，Florida Sparse Matrix Collection[44] より，表 4.1 に示す 10 個の疎行列を使用し実験を行った．この 10 個の疎行列は実際に FEM で使用された疎行列である．ここで N は疎行列の行数，N_z は疎行列内の非ゼロ要素数である．

表 4.1: Sparse matrices for preliminary expriment

Name of matrix	N	N_z
cant	62,451	4,007,383
rma10	46,835	2,374,001
consph	83,334	6,010,480
parabolic_fem	525,825	3,674,625
pwtk	217,918	11,524,432
thermal2	1,228,045	8,580,313
af_shell9	504,855	17,588,845
F1	343,791	26,837,113
nd24k	72,000	28,715,634
dielFilterV2real	1,157,456	48,538,952

本実験では，図 4.2 のように，疎行列内における各行中の非ゼロ要素の位置を，非ゼロ要素の連続数 (図 4.2 中の赤枠)，次の非ゼロ要素までのストライド (図 4.2 中の青枠) の並びでパターン化する．そして疎行列内に何個のパターンが存在するか（何個のパターンで疎行列中の全行を表すことができるか）調査する．多くの行のパターン同じであるほど，疎行列内のパターン数は少なくなる．

また疎行列内に存在する各パターンは，それぞれ何個の行で重複しているか調査する．疎行列内のパターンにそれぞれ Pattern number を割り振り，各 Pattern number のパターンで表される行が疎行列内に何行存在したかを示す．また，一致する行が 10 行以下のパターンに関しては，グラフの可読性向上のため，グラフ上には示していない．

表 4.2 に各疎行列中に存在するパターン数を示す．また図 4.3，図 4.4 に疎行列内に存在する各パターンは，それぞれ何個の行で重複しているかを示す．表 4.2 より 10 個中 8 個の疎行列において，疎行列の行数よりパターン数が少なくなっている．thermal2 はパターン数が少なくなっているが，その他の 7 個の疎行列に比べるとパターン数と行数の差は少ない．従来の疎行列格納方式ではこのパターンを考慮しておらず，同じパターンの行が存在した場合でも，全ての行の非ゼロ要素の位置情報を格納するため無駄が存在する．行数に対してパターン数が少ないほど，非ゼロ要素の位置情報を少ないデータ量で表すことが可能であり，メモリ使用量削減の効果が大きい．パターン数が行数より少なくなった疎行列を図 4.3，図 4.4 で確認すると，cant, consph, pwtk のように，少数のパターンに多くの行が重複していることが多い．ただし，af_shell9 や F1 のように少数のパターンに多くの行

表 4.2: Number of pattern in each sparse matrices

Name of matrix	N	Number of pattern
cant	62,451	114
rma10	46,835	14,243
consph	83,334	1,565
parabolic_fem	525,825	525,825
pwtk	217,918	4,361
thermal2	1,228,045	1,174,497
af_shell9	504,855	4,233
F1	343,791	120,456
nd24k	72,000	72,000
dielFilterV2real	1,157,456	578,406

が重複するのではなく，各パターンに満遍なく行が存在している疎行列も存在している．af_shell9 では 1000 個以上のパターンに対し，それぞれ多くの行が重複している．F1 では 10 個以上の行が重複しているパターンが 1 つしか存在しないが，非常に多くのパターンに対して 10 個以下の行が重複していることから，行数に対しパターン数が 20 万以上少ない結果となっている．

しかしながら parabolic_fem, (thermal2), nd24k, dielFilterV2real ではパターン数が行数に比べ多少少ないまたは同等である．このような疎行列において複数の行の非ゼロ要素の位置情報を少ないデータ量で表すことによるメモリ使用量削減の効果は希薄である．しかし，PatComp では連続した非ゼロ要素の列番号をどれだけ長い場合でも 1 つの連続数により表すことから，連続した非ゼロ要素の列番号に関するデータ量を削減する効果もある．そのため単純にパターン数が少ないためメモリ使用量が少なくならないと言い切ることはできない．

本予備実験では，複数の疎行列では全行の非ゼロ要素の位置関係を，行数に対し非常に少ないパターン数で表現できることを確認した．特に cant や consph，pwtk では行数に対してパターン数が 50 分の 1 以下であり，大幅な位置情報を表すデータ量を削減することが可能と考えられる．

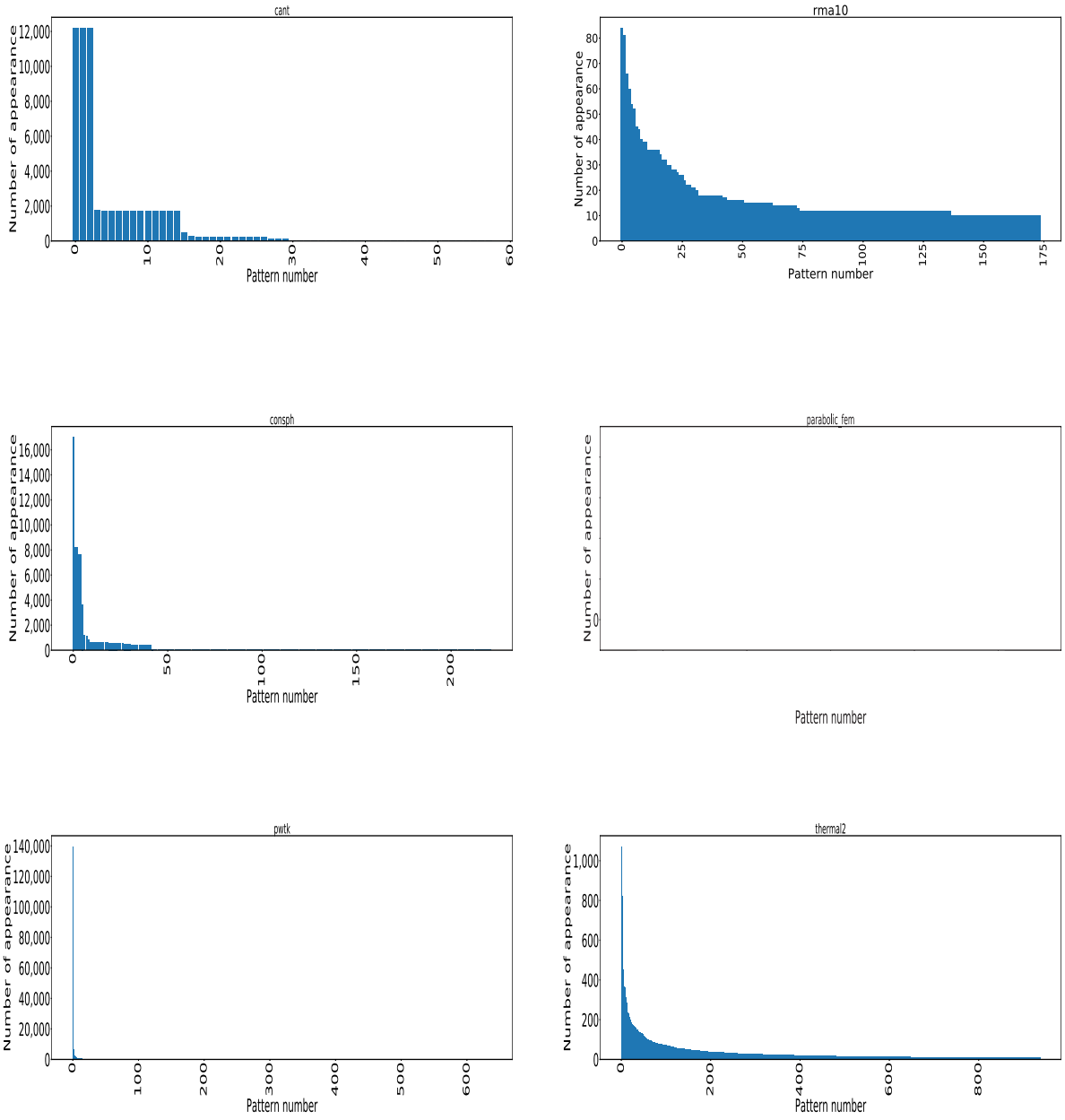

図 4.3: Number of appearance (cant to thermal2)

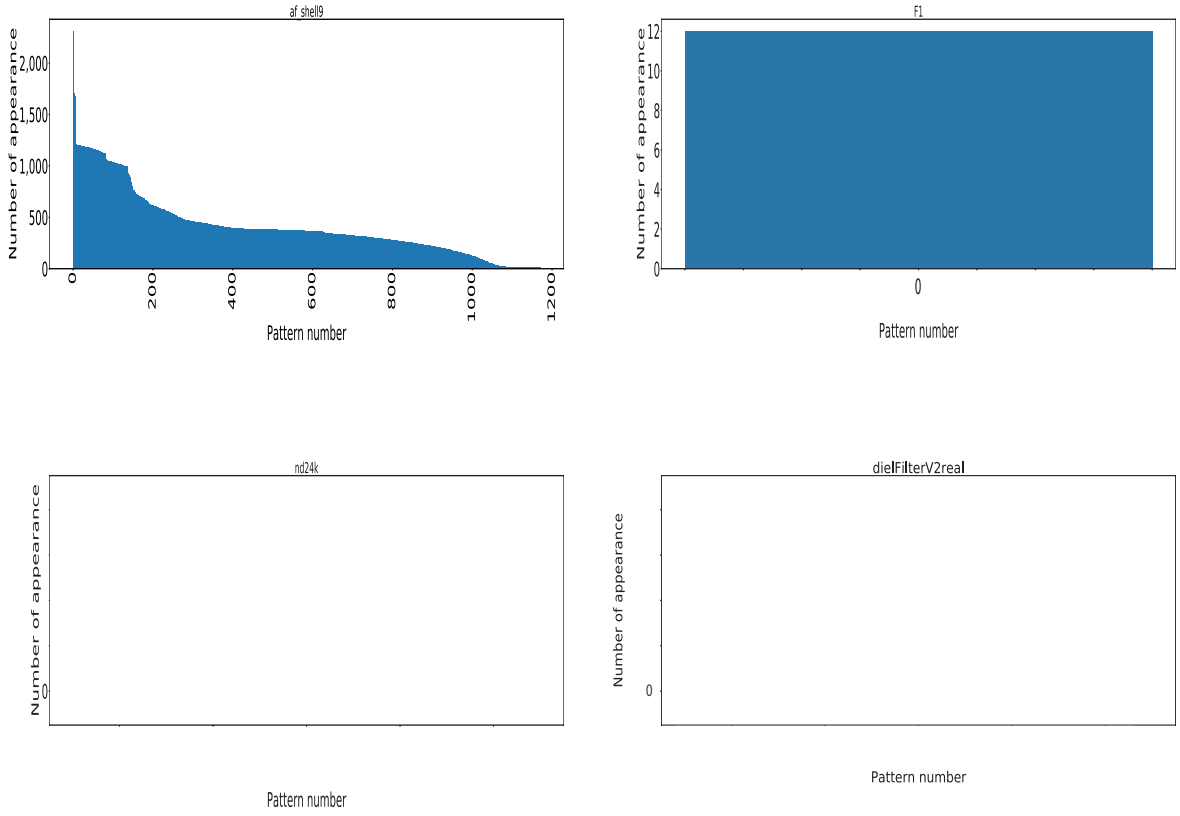

図 4.4: Number of appearance (af_shell9 to dielFilterV2real)

4.3 頻出するパターンを利用した既存データ圧縮方式と PatComp の差異

PatComp は疎行列の行を 1 行ずつ，非ゼロ要素の連続数と次の非ゼロ要素までの 0 の個数でパターン化し，辞書に登録する．その際，すでに存在辞書に登録されているパターンの場合には辞書に登録せず，辞書へのインデックスを格納する．このように 1 行の長い非ゼロ要素の位置関係を 1 つのパターンへの参照に置き換えることでメモリ使用量の削減を行う．これまでに既出のパターンを辞書に登録することでメモリ使用量を削減する手法は様々提案されている．例えば，LZ77 や LZ78，FVC も PatComp と同様に頻出する数列を辞書に格納し，メモリ使用量を削減する手法である．

PatComp と既存データ圧縮方式との差異は大きく分け，2 点存在する．まず 1 点目はパターン化の方法である．既存の LZ78 等では，1 つの同じ文字が複数回出現する長い配列（例えば CSR の *Columns*）に対して，頻出する数列を辞書に格納する．よって CSR の *Columns* 配列に対して LZ78 等の手法を適用することでメモリ使用量の削減は可能と

なる．しかし FEM で生成される疎行列では，各行の先頭の非ゼロ要素の位置が少しずつ右にずれている傾向にあることから，同じ列番号列が多く存在することは稀であると考える．それに比べ PatComp では FEM モデルの特性上，各行の先頭の非ゼロ要素の位置は違えど，そこからの非ゼロ要素の並びは同じであることが多いということを使用し，各行の非ゼロ要素の列番号を先述したパターンに置き換えている．このパターン化により，表 4.2 からも分かるように，疎行列内の非ゼロ要素の列番号を頻出するパターンとして見なすことができるようになる．よって PatComp のようなパターン化を採用することで，既存手法よりメモリ使用量削減の効果が大きくなると考えられる．また PatComp ではパターン化を行う際に，連続した非ゼロ要素の列番号を 1 つの連続数に置き換えている．この方法を採用することで，どれだけ長く連続した非ゼロ要素の塊があったとしても，列番号を示すために必要な要素数は 1 つとなり，メモリ使用量が大幅に削減される．FEM では連続した非ゼロ要素が非常に多く存在することからも，この連続数に置き換えるパターン化は有効であると考えられる．既存のパターンを使用するデータ圧縮手法として LZ78 を用いた CSR と PatComp のメモリ使用量の評価を 4.7.1 節にて行う．

2 点目はそれぞれの方式で疎行列を格納した際，GPU 上での SpMV において並列度が異なる点である．SpMV を GPU で高速に計算するためには，それぞれの方式で格納された疎行列を用いても，高い並列度で実行可能である必要がある．しかしながら LZ77，LZ78，FVC では圧縮されたデータを並列処理にて使用することは考えられておらず，SpMV において高い並列度を得ることができない．並列化に対応するためには，何かしらの工夫が必要となる．一方で PatComp では，1 行毎にパターン化を行うことで，辞書から読み出したパターンを GPU の各スレッドが並列して SpMV で使用でき，SpMV を行毎の高い並列度で実行可能である．また辞書型の圧縮方式では，元のデータを復元するために辞書へのアクセスが必要となるが，これは既存の疎行列格納方式に比べ，GPU の苦手なメモリアクセスの回数を増加させてしまう．そのためメモリアクセス回数増加により SpMV 演算速度が低下してしまう．しかし PatComp では連続した非ゼロ要素の列番号を 1 つの連続数で表していることから，SpMV の演算中は連続数分レジスタの値をインクリメントするだけで良い．よってメモリアクセス回数が減少し，辞書型圧縮方式の欠点であるメモリアクセス回数の増加を抑制する効果がある．

また PatComp を使用し Table を作成後に FPC を Table 内の整数データに適用しさらにメモリ使用量を削減することも考えられる．PatComp では非ゼロ要素の位置情報を表すために必要な要素数を削減することを目的としている．その後の Bit レベルでのメモリ使用量の削減は FPC 等の既存の手法を併用し行うことが可能である．以上が既存のパターンを用いるデータ圧縮方式と PatComp との差異である．

4.4 Pattern Compression (PatComp) 法

4.2.2 節において，少数のパターンで多くの疎行列内の非ゼロ要素の位置関係を表すことがメモリ使用量削減につながる可能性を示した．またパターンを作成する際に，連続

した非ゼロ要素の列番号は先頭のみを格納するため，連続した非ゼロ要素に対する圧縮の効果もある．また 1 行ごとにパターンの生成することで，SpMV において GPU の各スレッドは 1 行ずつ並列し計算が可能であり，高い並列度での実行が可能である．パターンを用いて疎行列内の非ゼロ要素の位置情報を削減するには，疎行列内の全パターンをテーブルに登録しておき，各行にインデックスを割り当てておく必要がある．各行のパターンがテーブルにすでに登録されているか確認するための処理が必要となることから変換時間は長くなる可能性がある．しかしながら時間発展に伴い FEM の格子が変わらず，疎行列の形が変わらない問題や収束までに多くの反復回数が必要となる問題では変換時間が全体処理時間の占める割合が少なくなることから，変換時間が長くてもメモリ使用量が少ない手法の有効性がある．時間ステップごとに疎行列の形が変わらない場合には，数値シミュレーションの最初に一度疎行列内の非ゼロ要素のパターンを登録しておくことで，その後の処理は値を変えるだけで処理が可能である．そこで本 4.4 章では，疎行列内のパターンを考慮した疎行列格納方式 Pattern Compression method (PatComp) を提案する．PatComp は既存手法に比べ，さらにメモリ使用量を削減することで，さらに大規模かつ高精度な数直シミュレーションを GPU 上で行えるようにすることが目的である．

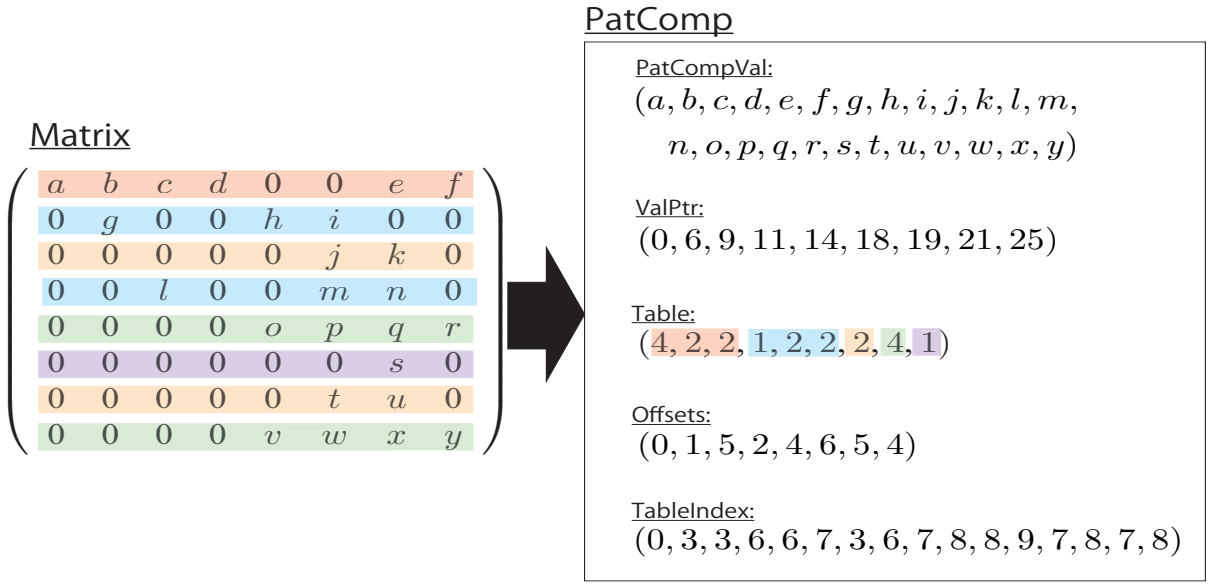

図 4.5: Pattern Compression method

図 4.5 に PatComp を用いた疎行列格納方法を示す．PatComp は，$PatCompVal$，$ValPtr$，$Table$，$Offsets$，$TableIndex$ の 5 つの配列を使用する．まず $PatCompVal$ に，各行の先頭の非ゼロ要素から順次値を格納する．$ValPtr$ は，疎行列内の各行の先頭の非ゼロ要素が $PatCompVal$ の何番目に格納されているか示す数値を格納する．図 4.5 では，0 行目の先頭の非ゼロ要素の値は $PatCompVal$ の 0 番目に格納されているため，$ValPtr$ の 0 番目に “0” を格納する．次に 1 行目の先頭の非ゼロ要素は，$PatCompVal$ の 6 番目に格納されているため，$ValPtr$ の 1 番目に “6” を格納する．

次に $Table$ へ，疎行列内に存在するパターンを 0 行目から順に追加する．図 4.5 の例では，0 行目のパターン (4, 2, 2)，1 行目のパターン (1, 2, 2)，2 行目のパターン (2) を順に追加している．但し追加する際，$Table$ 内に存在するパターンと重複する場合には，その行のパターンを追加しない．3 行目のパターンは (1, 2, 2) だが，1 行目と同じパターンであり，$Table$ 内のパターンと重複するため，追加していない．上記のようにパターンを追加していくことで，図 4.5 の例では 5 個のパターンが格納される．少ないパターンに多くの行が属するほど，非ゼロ要素の位置を表す情報を削減でき，$Table$ 内の要素数が少なくなるため，メモリ使用量が大きく削減される．

$Offsets$ は，各行のパターンが何列目から始まるかを格納する配列である．$Offsets$ の 0 番目の要素は，図 4.5 の 0 行目に対応しており，0 列目から (4, 2, 2) の並びで非ゼロ要素が存在することを示す．同様に $Offsets$ の 1 番目の要素から，疎行列の 1 行目は，1 列目から (1, 2, 2) の並びで非ゼロ要素が存在することを示す．

最後に $TableIndex$ は，各行のパターンを $Table$ 内の何番目から何番目の要素が表しているかを格納するための配列である．疎行列内の i 行目のパターンは，$TableIndex$ 内の “$2*i$” 番目と “$2*i+1$” 番目の要素を使用し，$Table$ へアクセスすることで得ることができる．例えば 0 行目のパターンは，$TableIndex$ 内の 0 番目 (“0”) と 1 番目の要素 (“3”) から，$Table$ 内の 0 から 3 未満の (4, 2, 2) のように判断する．1 行目のパターンは，$TableIndex$ 内の 2 番目 (“3”) と 3 番目の要素 (“6”) から，$Table$ 内の (1, 2, 2) であることが分かる．

以上が PatComp を用いた疎行列の格納方法である．PatComp は連続した非ゼロ要素の先頭の列番号のみ格納することから，連続した非ゼロ要素の列番号数削減効果もある．また 1 つのパターンを複数の行で使い回すことから Cache hit rate の向上も期待できる．

FEM を使用した数値シミュレーションを行う際，連立一次方程式を表す疎行列は COO 等の単純な疎行列格納方式で表されることが一般的である．そのため PatComp は FEM モデルから生成され，COO 形式で格納されている疎行列を PatComp へ変換し，反復法内の SpMV で使用することを想定する．そのため Algorithm6 に，COO 形式で格納された疎行列を PatComp へ変換するための疑似コードを示す．Algorithm 中の $Offsets$，$PatCompVal$，$ValPtr$，$Table$，$TableIndex$ は，図 4.5 内の配列に対応する．また $Columns$，$Values$，Row は図 3.7 に示す COO の配列である．

$PreCol$ は一つ前の非ゼロ要素の列番号を格納しておくための変数である．1 行目では $Offsets$ へ 0 行目の先頭の非ゼロ要素の列番号を格納している．$CurrentRow$ は現在扱っている非ゼロ要素が何行目の非ゼロ要素かを示す．

4 行目の for 文で使用する i は疎行列内の何番目の非ゼロ要素かを示す．ここで N_z は疎行列内の非ゼロ要素数を示す．$PatCompVal$ は COO の $Values$ と同じ配列になるため，5 行目で反復毎に $Colmuns$ の値を $PatCompVal$ へ代入する．6 行目から 21 行目までは，i 番目に格納されている非ゼロ要素が各行の先頭の非ゼロ要素である場合の処理である．行が変わった際には $Offsets$ に先頭の非ゼロ要素の列番号を格納 (7 行目) し，$ValPtr$ へ $PatCompVal$ の i 番目の要素が行の先頭であることを記録する必要がある．また前行の非ゼロ要素のパターンがすでに $Table$ が登録されているかをチェックし，されていない場

合にはパターンを $Table$ に追加する必要がある．ここで $RowPattern$ は前の行の非ゼロ要素のパターンを記憶しておくための 1 次元配列である．$RowPattern$ には非ゼロ要素の連続数と次の非ゼロ要素までのストライドが交互に格納される．

10 行目から 15 行目までが $RowPattern$ で示されるパターンが $Table$ に登録されていない場合の処理である．13 行目において $Table$ へ $RowPattern$ に格納されているパターンを順に代入していく．そして $Tabel$ へ格納されたパターンの先頭と末尾のインデックスを 11 行目と 15 行目で $TableIndex$ に格納し，前の行が $Table$ 内のどのパターンであるかを記憶する．

16 行目から 19 行目はすでに $RowPattern$ が $Table$ に登録されていた場合の処理である．登録されている場合には，$RowPattern$ が $Table$ 内のどの位置に格納されているか判別するため，$TableIndex$ へパターンが登録されている先頭と末尾のインデックスを追加する (17，18 行目)．

10 行目，17 行目，18 行目の波線部に示す，あるパターンが $Table$ を登録されているか確認し，登録されている場合は登録されている位置を返す処理は，$Table$ に登録されている多くのパターンと比較する必要があるため，計算量が非常に大きくなる．しかし比較を行う処理は並列して行うことができるため，実際の実装では OpenMP による並列化を行い，高速化を図っている．

前の行のパターンに関する処理が終了後，20 行目で $RowPattern$ を初期化する．21 行目の $TmpCnt$ は i 番目の非ゼロ要素が何連続目の非ゼロ要素かを示す．21 行目では扱う i 番目の非ゼロ要素は，行が変わって最初の非ゼロ要素でことから 1 が代入される．

22 行目から 30 行目のは i 番目の非ゼロ要素が行の先頭以外の要素に対する処理である．前の非ゼロ要素の列番号を示す $PreCol$ と $Columns[i]$ の差が 1 であるとき，二つの非ゼロ要素は連続しているため，連続数を示す $TmpCnt$ をインクリメントする (23，24 行目)．二つの非ゼロ要素が不連続な場合，その時点での $TmpCnt$ を $RowPattern$ に格納後，二つの非ゼロ要素のストライドを計算し，$RowPattern$ へ格納する必要がある．そして i 番目の非ゼロ要素が前の要素と連続していないことから，$TmpCnt$ に 1 を代入し，連続数をリセットする．その後 31 行目にて，i 番目の非ゼロ要素の列番号を $PreCol$ に代入する．4 行目から 32 行目の処理を COO 内の全ての非ゼロ要素に対し行った後，33 行目にて $ValPtr$ へ非ゼロ要素数を示す N_z を格納し終了となる．

Algorithm 6 Matrix compression of PatComp

```
1: PreCol = Offsets[0] = Colums[0]
2: CurrentRow = Row[0]
3: PatCompVal = Values[0]
4: for i = 1 to N_z do
5:    PatCompVal[i] = Values[i]
6:    if CurrentRow! = Row[i] then
7:       Offsets[OffsetCnt++] = Columns[i]
8:       ValPtr[PtrCnt++] = i
9:       RowPattern[RowPatCnt++] = TmpCnt
10:      if "RowPattern not in Table" then
11:         TableIndex[IndexCnt++] = TableCnt
12:         for  j = 0 to RowPatCnt do
13:            Table[TableCnt++] = RowPattern[j]
14:         end for
15:         TableIndex[IndexCnt++] = TableCnt
16:      else
17:         TableIndex[IndexCnt++] = "Head index which stored pattern as same as
            RowPattern in Table"
18:         TableIndex[IndexCnt++] = "Tail index which stored pattern as same as
            RowPattern in Table"
19:      end if
20:      "Initializing RowPattern"
21:      TmpCnt = 1
22:   else
23:      if Columns[i] − PreCol == 1 then
24:         TmpCnt = TmpCnt + 1
25:      else
26:         RowPattern[RowPatCnt++] = TmpCnt
27:         RowPattern[RowPatCnt++] = Columns[i] − PreCol
28:         TmpCnt = 1
29:      end if
30:   end if
31:   PreCol = Columns[i]
32: end for
33: ValPtr[PtrCnt] = N_z
```

式 (4.1) に PatComp を用いて疎行列を格納した場合のメモリ使用量を示す．ここで，N_T は $Table$ 内の要素数を表す．PatComp に必要となる配列の要素数は，$PatCompVal$ が N_z，$ValPtr$ が $N+1$，$Offsets$ が N，そして $TableIndex$ が $2N$ である．$PatCompVal$ が倍精度浮動小数点数，それ以外が非負整数であることから，式 (4.1) が導出される．

$$\begin{aligned} MemUsage_{PatComp} &= 8N_z + 4(N+1) + 4N_T + 4N + 4*2N \\ &= 8N_z + 4N_T + 16N + 4\,[byte] \end{aligned} \tag{4.1}$$

4.5 PatCompを用いたSpMVカーネル

本4.5節にて，PatCompで格納された疎行列を用いたSpMV演算の説明を行う．Algorithm7にPatCompを用いたSpMVコードを示す．Algorithm7は，GPU上の各スレッドで並列に実行される．1行目，iは実行するスレッド番号を表し，疎行列のi行目のSpMV演算を担当する．Algorithm中の$TmpResult$は，SpMV演算の途中結果を格納しておく変数である．3行目にて，i行目の先頭の非ゼロ要素の位置を$Offsets[i]$から読み込む．次に4行目，5行目にて$Table$からi行目のパターンを読み出すため，$Table$中のi行目のパターンの先頭位置$Head$と，そこから何個がi行目のパターンかを示す$Diff$を用意する．6行目の$ValHead$は，i行目の先頭の非ゼロ要素の値が$ValPatComp$の何番目の要素かを示す．

7行目から15行目がi行目の非ゼロ要素に対するSpMV演算を行うループを表す．パターンの先頭は必ず非ゼロ要素の連続数であるため，7行目から9行目にて，ColNumから連続数分，列番号をインクリメントし，SpMV演算を行う．レジスタに格納されている列番号をインクリメントするだけで本来の列番号が復元できることから，従来のCSR等の既存手法では必要であったメモリアクセスが削減され，演算性能向上の効果がある．最初に$OffCol$には，各行の先頭の非ゼロ要素の列番号が格納されているため，$OffCol$を非ゼロ要素の連続数分インクリメントしていくことで，次の乗算に使用するベクトル$dVector$のインデックスを計算することが可能である．また，$ValHead$は$ValPatComp$から読み出す非ゼロ要素の値の位置を表しているため，1回のSpMV計算をするたびにインクリメントし，次のSpMVで使用する値の位置を保持する．その後の10行目から15行目では，7行目から9行目で使用しなかったパターンの要素を用いて，SpMV演算を行う．11行目にて$ColNum$に次の非ゼロ要素までの0の数を足すことで，次の非ゼロ要素の列番号を求める．12行目から14行目では7行目から9行目と同様に，非ゼロ要素の連続数を用いてSpMV計算を行う．

最後に16行目で，SpMVの演算結果$TmpResult$を演算結果を格納する配列Ansへ格納する．

4.6 PatComp評価実験概要・環境

PatCompは既存の疎行列格納方式の高い並列度を保ちながら，既存の疎行列格納方式より少ないメモリ使用量で疎行列を格納することが目的である．PatCompの目的が達成されているか確認するため，評価実験を行う．評価実験ではPatCompを用いて疎行列を格納した際のメモリ使用量，SpMV演算時間を評価する．またPatCompは疎行列からの変換に多くの計算を必要とするため，疎行列（COO）からPatCompへの変換時間が反復法，GMRESの演算時間に与える影響を評価するため，GMRESを用いた評価を行う．

表4.3に本稿の評価実験で用いたGPUサーバの実験環境を示す．本評価実験で使用した15個の疎行列を表4.4に示す．上から13個は，従来手法であるCSRやELL，ELL-R

Algorithm 7 SpMV code of PatComp on GPU

1: Let i be thread ID (id: 0 to n-1)
2: $TmpResult = 0$
3: $OffCol = Offsets[i]$
4: $Head = TableIndex[2 * i]$
5: $Diff = TableIndex[2 * i + 1] - Head$
6: $ValHead = ValPtr[i]$
7: **for** $k = 0$ to $Table[Head]$ **do**
8: $\quad TmpResult = TmpResult + ValPatComp[ValHead++] * dVector[OffCol++]$
9: **end for**
10: **for** $j = 1$ to $Diff$ **do**
11: $\quad OffCol = OffCol + Table[Head + j] - 1$
12: $\quad$ **for** $k = 0$ to $Table[Head + j + 1]$ **do**
13: $\quad\quad TmpResult = TmpResult + ValPatComp[ValHead++] * dVector[OffCol++]$
14: $\quad$ **end for**
15: **end for**
16: $Ans[i] = TmpResult$

と比較するため Florida Sparse Matrix Collection[44] 収録の行列で評価した．この中で比較的大規模な行列として DielFilterV2read (2011) や Cube_Coup_dt0 (2012), Queen_4147 (2014), Bump_2911 (2014) がある．2011 年当時に利用可能な最大の GPU は Tesla M2090 で 6GB，同じく 2014 年では K40 が 12GB であるが，上記行列は従来手法で圧縮してもこれらの GPU のほぼ上限のサイズである．提案手法によって圧縮率が向上すれば，より安価で小容量の GPU が利用できるか，またはシミュレーションサイズを拡大することが可能である．UT-Heart1，UT-Heart2 は，心臓シミュレーション [8] の流体構造連成解析における，非圧縮性を扱った混合型の定式化により現れる行列である．医療分野のアプリケーション例として評価対象に入れた．使用した疎行列は COO 形式でファイルに保存されており，プログラム中でこの疎行列ファイルを読み込みを行った後，COO からそれぞれの疎行列へ変換する．

また，表 5.2 中の N,N_z,K，N_T は，式 (3.1)，式 (3.2)，式 (3.3)，式 (4.1) で使用されている変数に対応している．そして N_{non} は疎行列内の不連続な非ゼロ要素の数を示しており，参考のため表に示している．本実験にて使用する疎行列格納方式は，CSR, ELL, ELL-R, PatComp である．

図 4.6 は実験で使用する，FEM にて生成された連立一次方程式（疎行列）に PatComp 法を適用し，GMRES で解くことを想定したモデルを示す．赤枠より上の処理，「FEM モデルの生成」,「作成された疎行列をファイルに保存」で保存された疎行列に，Florida Sparse Matrix Collection またはアプリケーション例として加えた 2 つの疎行列を使用することで，評価を行う．評価対象は，図 4.6 中の赤枠，つまり疎行列格納方式への変換か

表 4.3: Expriment condition

	Specification
OS	Ubuntu 16.04
CPU	Intel Core i7 6700K @ 4.0 GHz
GPU	NVIDIA Tesla V100 @ 1.38 GHz
device memory	16 GB
device memory bandwidth	900 GB/s
Number of CUDA core	5,120
CUDA version	CUDA 10.1
C Compiler	gcc-4.4.7

らGMRES処理の終了までである．GPUによる評価を行うために，NVIDIAが提供するCUDA10.1を使用し，各疎行列格納方式を用いたSpMVを組み込んだGMRESプログラムを記述した．疎行列ファイルの読み込みからPatCompの適用，GMRESの初期処理までがCPU上で処理し，その後はGPU上での処理となる．図4.6の青枠で囲まれた部分がGPU上で実行される．データ転送時間，GPU演算時間は，CUDAのイベント変数を用いて，“cudaEventRecord()”,“cudaEventElapsedTime()” により測定を行った．

4.7節では，PatComp法を用いて疎行列を格納した際に必要となるメモリ使用量と図4.6内のSpMV(赤ブロック)のSpMVを1回実行した際の演算時間を評価する．最後に5.9節にて，図4.6内の赤枠全体の演算時間の評価を行う．

4.7 PatCompの性能評価実験結果

PatComp法は疎行列内の非ゼロ要素にパターン性が存在することに着目し，複数の行で共通のパターンを格納しておくことで，多くの行の非ゼロ要素の位置を少ないパターン数で表現する手法である．多くの行の非ゼロ要素の位置を少ないパターンで表現するほどメモリ使用量削減の効果が高くなる．また連続した非ゼロ要素の列番号を少ない要素数に置き換えるため，連続した非ゼロ要素が疎行列中に多く存在する場合もメモリ使用量削減の効果が高くなる．4.7.1にてPatComp法を用いた際のメモリ使用量を評価する．CSRの少ないメモリ使用量で疎行列を表現することを目指した手法と比較しても，PatComp法のメモリ使用量の方が少なくなっている場合には，パターン性を考慮する効果が非常に高いことが確認できる．

表 4.4: Sparse Matrices for the expiments

Name of matrix	N	N_{z}	N_{non}	K	N_{T}
cant	62,451	4,007,383	87,657	78	1,860
rma10	46,835	2,374,001	447	145	163,463
consph	83,334	6,010,480	52,325	81	46,517
parabolic_fem	525,825	3,674,625	2,081,987	7	5,034,181
pwtk	217,918	11,524,432	4,049	180	42,267
thermal2	1,228,045	8,580,313	5,192,466	11	11,533,043
af_shell9	504,855	17,588,845	0	40	22,213
F1	343,791	26,837,113	9,300	435	3,878,118
nd24k	72,000	28,715,634	887,233	520	6,714,354
dielFilterV2real	1,157,456	48,538,952	18,745,748	110	27,351,242
Cube_Coup_dt0	2,164,760	124,406,070	0	68	5,773,972
Bump_2911	2,911,419	127,729,899	86,208	195	10,972,937
Queen_4147	4,147,110	316,548,962	1	81	16,904,718
UT-Heart1	82,047	3,423,519	0	63	501,899
UT-Heart2	130,595	6,954,413	969,595	124	1,242,317

4.7.1 各疎行列格納方式のメモリ使用量の評価

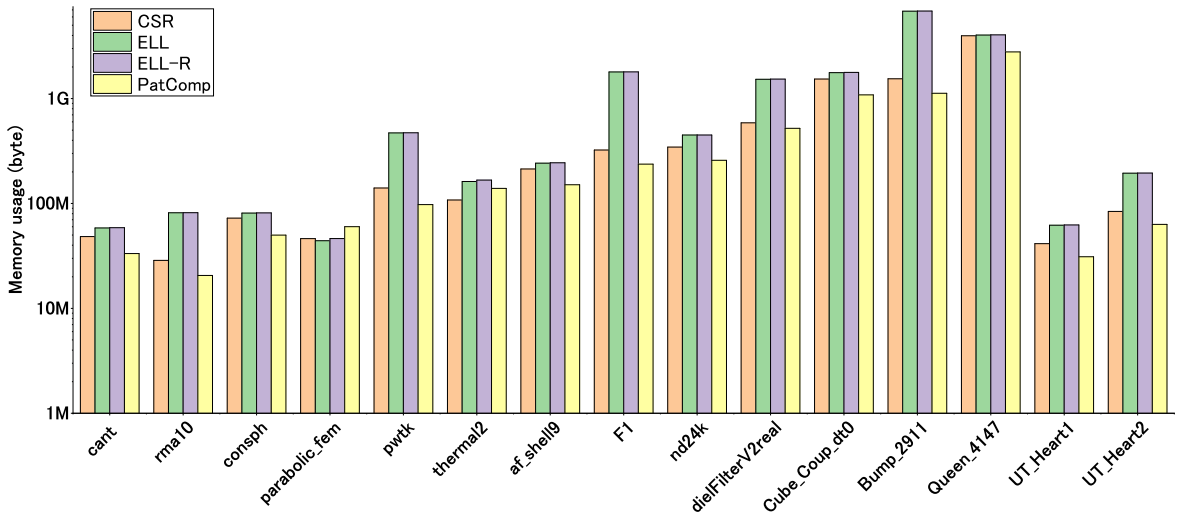

図 4.7: Memory usage of PatComp format

本評価実験では ELL の派生系である ELL-R も比較対象として，評価結果を示している．

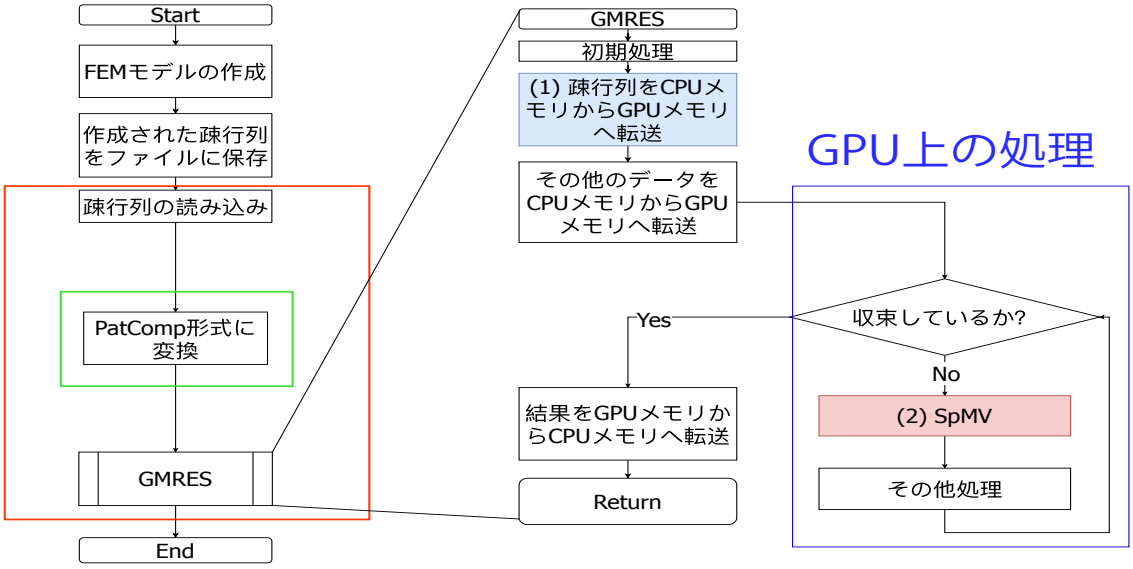

図 4.6: Model of Calculating GMRES with PatComp

図 4.7 に PatComp を含む，各疎行列格納方式のメモリ使用量を示す．15 個中 13 個の疎行列において PatComp のメモリ使用量が最小となった．

CSR と比較すると平均 19.4%，cant と consph において最大 31.1%のメモリ使用量の削減に成功した．CSR を用いて consph を格納した場合のメモリ使用量のうち，66.4%が $Values$ のメモリ使用量である．よってその他の非ゼロ要素の位置を表す情報を格納する $Columns$ と $RowPtr$ を全て削除しても，メモリ使用量の削減率は 33.6%である．PatComp の削減率 31.1%は 33.6%に非常に近い値であり，大幅に位置情報に必要なメモリ使用量を削減していることが分かる．また 15 個中 4 個の疎行列において 30%以上，12 個の疎行列において 25%以上のメモリ使用量削減を達成した．

疎行列内の非ゼロ要素の連続性のみを考慮した RBP-CSR に比べ，平均で 4.3%，UT_Heart2 において最大 14.1%のメモリ使用量を削減した．この結果から非ゼロ要素の連続性のみでなく，パターン性も考慮することでメモリ使用量をさらに削減できることを示した．予備実験で使用していない FEM により生成された疎行列においても PatComp の効果があったことから，多くの疎行列は少数のパターンで表現でき，PatComp によるメモリ使用量削減の効果があると考えられる．

しかしながら，“parabolic_fem”，“thermal2” ではメモリ使用量の削減を達成できなかった．parabolic_fem，thermal2 では，連続した非ゼロ要素が 1 個しか存在しない行が多く，規則性，パターン性を考慮しても，メモリ使用量削減の効果が得られなかった．また不連続な非ゼロ要素が多く存在し，PatComp で格納する場合には 1 つの非ゼロ要素に対し CSR より多くの値を用いることから CSR より大幅にメモリ使用量が増加する原因となった．今後，不連続な非ゼロ要素に関しては CSR で格納するなどの対策を講じることを課

題とする．

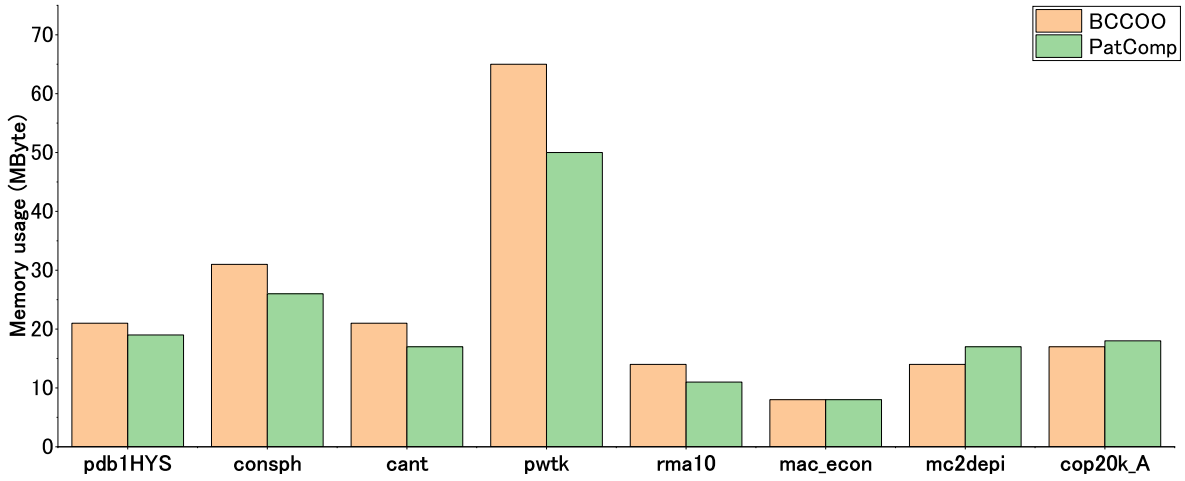

図 4.8: Comparision of memory usage with BCCOO

また既存手法との比較として BCCOO と PatComp のメモリ使用量を図 4.8 に示す．BCCOO のメモリ使用量は [41] に記載されているメモリ使用量を使用した．また比較のため，[41] で使用されている 7 個の疎行列に対して PatComp による変換を行い，PatComp のメモリ使用量を測定した．BCCOO は著者の知る中で最もメモリ使用量の低い疎行列格納方式の一つである．PatComp と BCCOO のメモリ使用量を比較すると，平均で 7.7%，最大 23.1%のメモリ使用量の削減に成功した．疎行列のパターン性を考慮することで BCCOO よりメモリ使用量を削減できたことから，パターン性を考慮することのメモリ使用量削減の有効性を示した．

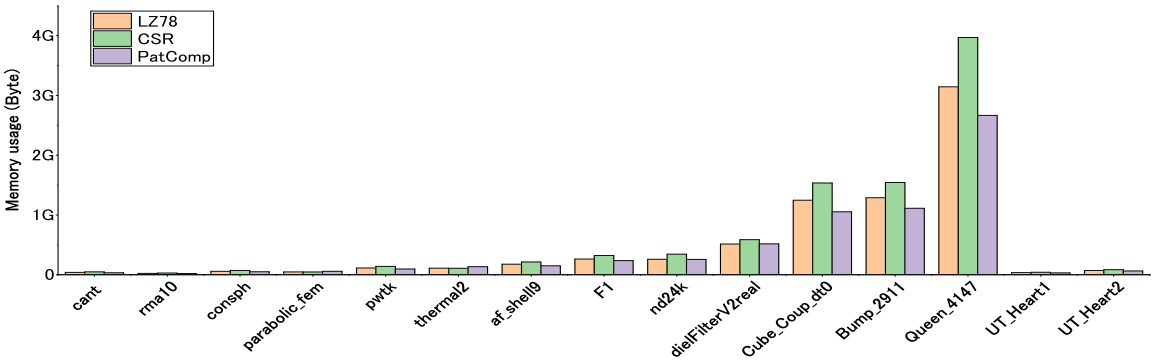

図 4.9: Comparision of memory usage with LZ78

最後に既存のパターンを使用するデータ圧縮方式との比較を行う．従来の頻出するパ

ターンを再利用しメモリ使用量を削減する手法（LZ77, LZ78, FVC 等）では，疎行列の列番号を 1 つの数列とみなした時（例えば CSR の $Columns$），頻出する数列が多く存在しないことからメモリ使用量削減の効果が希薄であることが問題であった．PatComp は連続数と次の非ゼロ要素までの 0 の個数を使用し，各行の非ゼロ要素の位置をパターン化している．このようなパターン化を行うことで，通常での頻発しないパターンが頻発するようになり，既存の辞書等を用いたデータ圧縮方式より大幅なメモリ使用量の削減が可能である．PatComp のパターン化方法の有効性を検証するため，LZ78 を CSR の $Columns$ 配列に適用した場合のメモリ使用量（CSR の $Values$, $RowPtr$ と LZ78 を適用した $Columns$ の合計）と PatComp のメモリ使用量を比較する．ただし LZ78 を CSR の $Columns$ に適用した場合，LZ78 が先頭からの復元を必要とする特徴があるため，GPU による並列処理は不能となる．今回はメモリ使用量の単純な比較のため，LZ78 を CSR の $Columns$ に適用した方式を使用する．並列処理を可能とするには，さらに Ptr 配列などの追加による工夫が必要であると考えられる．また FVC も LZ78 と同様に頻出する数列を辞書に格納し，メモリ使用量を削減する手法である．しかし FVC はキャッシュ内の短いデータ列を対象にしているのに対し，疎行列の配列は非常に長い．そのため FVC を用いた際のメモリ使用量は，長いデータ列になるほどメモリ使用量削減率が高い LZ78 より大きくなると考えられる．そのため今回は LZ78 と PatComp のメモリ使用量の比較を行う．

図 4.9 に PatComp と LZ78 を施した CSR(図中 LZ78) のメモリ使用量を示す．parabolic_fem, thermal2, dielFilterV2real 以外の 12 個の疎行列において PatComp のメモリ使用量が少なくなっている．PatComp のメモリ使用量は LZ78 と比べ，平均 5.6%，最大 16.4%メモリ使用量が少なくなった．この結果から PatComp は GPU での高い並列度を保ちながらも，FEM で生成される疎行列の特性を使用しパターン化を行うことで，既存のパターンを用いる手法に比べメモリ使用量が少なくなった．LZ78 を用いて GPU 上での SpMV を実行するためにはさらに配列を追加する必要があるため，LZ78 を用いた際のメモリ使用量はさらに増加する．

PatComp を使用することでメモリ使用量が増加した parabolic_fem, thermal2 は，LZ78 においても CSR のメモリ使用量に比べ増加していることから，これらの疎行列に頻出する数列が非常に少ないと考えられる．また dielFilterV2real では LZ78 のメモリ使用量は CSR，PatComp のメモリ使用量より少なくなっている．dielFilterV2real は 1 行単位でのパターン化が効果的でなく，LZ78 のように行中の数列でも辞書に登録するほうがメモリ使用量が少なくなる例である．今後このような問題においてもメモリ使用量の削減が可能なよう PatComp の改良が必要となる．

しかしながら多くの疎行列において PatComp は既存のパターンを使用する疎行列格納方式に比べメモリ使用量が少なくなると考えられ，PatComp のパターン化方法がメモリ使用量の削減に効果的であることを示した．

4.7.2 GPU 上での SpMV 演算時間の評価

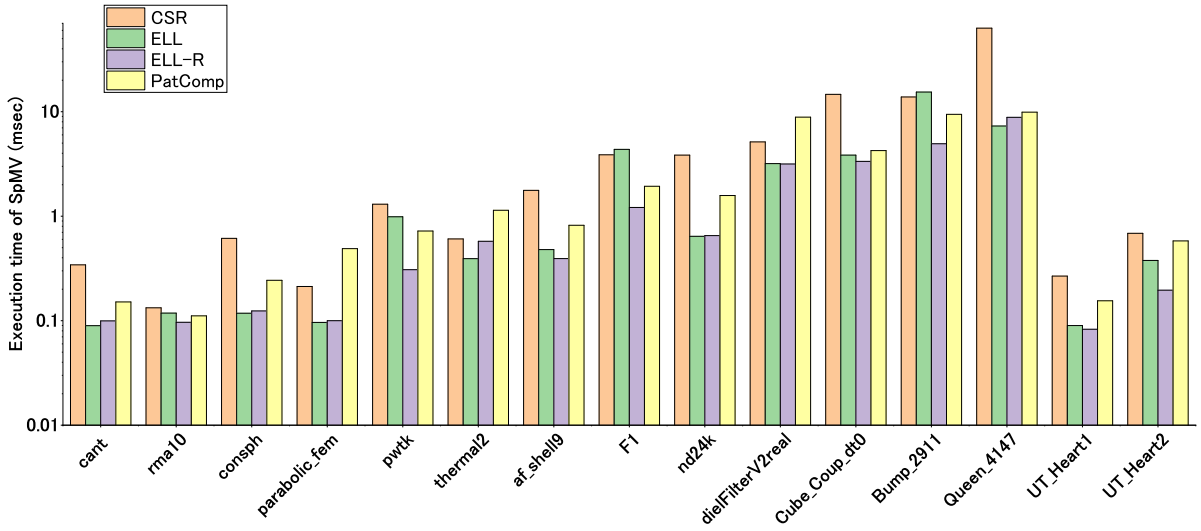

図 4.10: Execution time of SpMV using PatComp format on GPU

前節では PatComp が多くの疎行列で，CSR よりメモリ使用量が少ない疎行列格納方式であることを明らかにした．本節では PatComp の SpMV 演算性能が CSR の SpMV 演算時間の比較をして，遜色ないことを確認することが目的である．

図 4.10 に PatComp を用いた GPU 上での SpMV 演算時間を示す．CSR を用いた SpMV 演算時間と比較し，PatComp を用いた SpMV は 15 個中 12 個の疎行列にて演算時間が短くなった．平均で 1.85 倍，最大 3.45 倍の高速化に成功した．PatComp は連続した非ゼロ要素の連続数を 1 つの値で表しており，レジスタの値をインクリメントするだけで，元々の列番号を復元することが可能である．そのためメモリアクセス回数の削減の効果があり，これが SpMV 演算時間短縮に繋がったと考えられる．また複数の行が同じパターンに属する場合には，各 Thread が同じデータを $Table$ から読み出すため，パターンの再利用性が高まり，GPU 上のキャッシュを効率的に使用できたと推測する．この 2 点が PatComp の演算時間短縮の要因である．メモリ使用量の削減に成功した疎行列において，演算時間の削減率が大きくなる傾向がある．

しかしながら，CSR と比較して演算時間が長くなる疎行列も存在する．parabolic_fem，Thermal2，nd24k にて PatComp の SpMV 演算時間が伸び，この 3 つの疎行列は PatComp を用いることでメモリ使用量が増加した疎行列である．これは RBP 法より多くの配列または行列へのアクセスが発生するためと考えられる．parabolic_fem，thermal2 は不連続な非ゼロ要素を表現するために必要となる値の数が CSR に比べ PatComp では増加する．そのためその増加した値に対してのメモリアクセスを行う必要があり，SpMV 演算時間が長くなったと考えられる．nd24k については不連続な非ゼロ要素は 2 つの疎行列に比べると少ないが，少量のパターンで多くの行を表すことができず，$Table$ へのアクセス時間が

増加したことが原因であると考えている．よって PatComp により SpMV 演算時間の向上が見られる疎行列の特徴として，不連続な非ゼロ要素数が少ないこと，そして多くの行を少ないパターンで表現できることの 2 つが挙げられる．

4.8 GMRES による演算時間の評価

4.8.1 各疎行列格納方式を用いた GMRES の演算時間

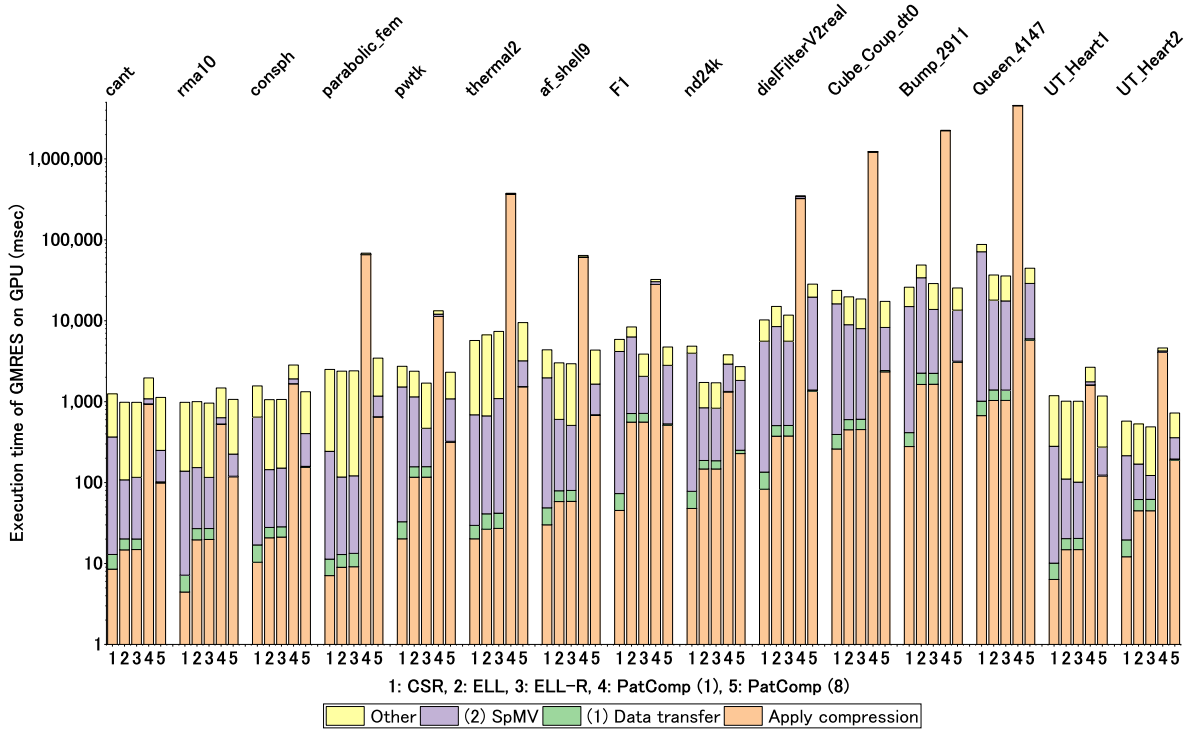

図 4.11: Execution time of GMRES on GPU

図 5.15 に，連立一次方程式求解の処理時間を評価するため，Algorithm1 に示す GMRES 終了までの CPU 処理を含めた処理時間を示す．縦軸は対数軸となっている．内訳として，図 5.5 における (1) 疎行列の CPU メモリから GPU メモリへの転送時間，(2)GPU 上での SpMV 演算時間の合計，およびその他の処理時間を，それぞれ緑，紫，黄色で表した．その他の処理時間 (Other) は、GMRES(図 5.5 の赤枠部分) から各処理時間を除いた残りの時間である．また疎行列から各疎行列格納方式への変化時間をオレンジ色で示す．データ転送時間は各疎行列格納方式で格納された疎行列の転送時間のみを示している．行列本体以外のデータは疎行列格納方式に関わらず一定であるため，その他の処理時間に含めている．今回，GMRES の最大反復回数は，最大 1000 回としている．数値シミュレーショ

ンの必要とする精度等により，必要な反復回数が異なるため，今回は反復回数 1000 回での GMRES 演算時間を評価する．例外として UT_Heart2 は 277 回の反復で収束したため，その時点での GMRES 演算時間を示す．

連立一次方程式の求解中で，疎行列のデータ転送は 1 回のみであり，SpMV の実行は反復毎に行われるため，全体の処理時間に対する SpMV 演算時間の割合が，図 5.15 から多くの疎行列において高くなっていることが確認できる．

変換時間を含まない PatComp の GMRES 演算時間と CSR の GMRES 演算時間を比較すると，PatComp の GMRES 演算時間の方が平均で 4.3%短くなっている．

PatComp のデータ転送時間は CSR のデータ転送時間と比べ，平均で 19.2%短くなっている．pwtk を用いた際，最も短縮した時間が大きく，31.9%のデータ転送時間短縮に成功した．この結果から PatComp によるメモリ使用量削減に伴いデータ転送時間も削減されたことを確認した．しかしメモリ使用量削減が困難であった parabolic_fem，thermal2 においてはデータ転送時間の増加が確認された．

しかしながら COO から PatComp への変換時間を含めると多くの疎行列において PatComp を用いた GMRES の演算時間が長くなっている．PatComp への変換を逐次的に行った場合の GMRES の演算時間が “PatComp(1)”，8 スレッドで並列化した場合の GMRES の演算時間が “PatComp(8)” としてグラフに示している．PatComp は数値シミュレーション中に解析対象の物体が破壊や亀裂等により FEM モデルが変化せず，メッシュの形状の変化もない問題を対象としている．時間発展毎にメッシュの形状に変化がしない場合，反復法中で使用する疎行列の位置にも変化がないそのため数値シミュレーション中で PatComp への変換が行われる回数は最初の 1 回であり，多くの変換時間が必要となったとしても，ステップが進むごとに変換時間の占める割合は相対的に減少する．しかし，時間ステップごとに疎行列の形が変わる問題では，ステップごとに変換が必要となることから変換時間が長い PatComp は不向きである．

4.8.2 COO 形式から各疎行列格納方式への変換時間

本節では PatComp の変換時間のみに注目して評価を行う．図 5.5 中の緑色の枠で示した COO 形式から各疎行列格納方式への変換時間のみを，図 5.17 に示す．COO から各疎行列格納方式への変換は CPU 上で行われる．PatComp の変換時間は，COO から PatComp への変換に要した時間である．

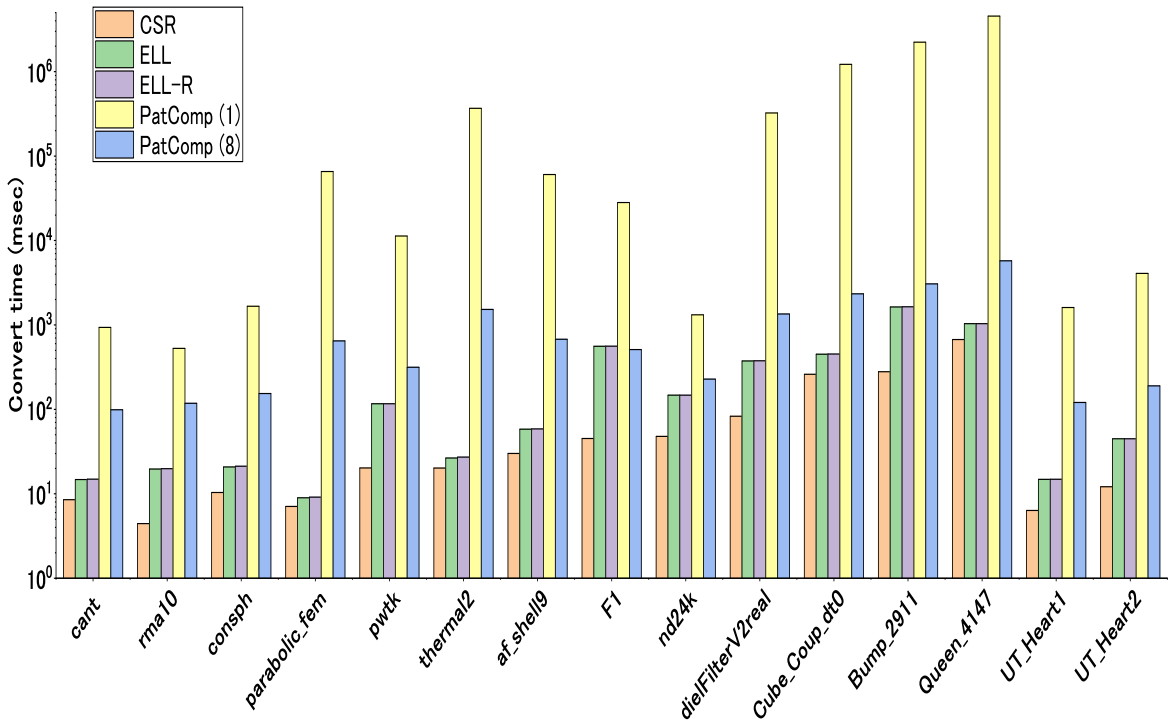

図 4.12: Convert time of each formats

、

図 5.17 に示す PatComp(1) は，Algorithm6 中の波線部の処理を逐次的に行った場合の変換時間を示している．波線部の処理は，疎行列中の各行のパターンが $Table$ に登録されているか確認し，登録されている場合には $Table$ 中のそのパターンが存在する位置を返す処理である．各行のパターンを $Table$ に登録されているパターンと比較する必要があるため，計算量が非常に大きくなる．そのため逐次的に COO から PatComp へ変換した場合，他の疎行列に比べ非常に変換時間が長くなっている．特にメモリ使用量を削減できなかった疎行列 parabolic_fem，thermal2 では，非ゼロ要素数が比較的少ないながらも，変換時間が非常に長くなっている．これはパターンが重複する行が少ない場合，$Table$ 内のパターン数が多くなり，各行のパターンが $Table$ 内に存在するか確認する処理の計算量が大きくなるためである．

しかしながら波線部は並列して処理することが可能であるため，OpenMP にて並列化することにより，PatComp への変換の高速化を図っている．登録されているか確認したいパターンと $Table$ に登録されているパターンとの比較を並列して行うことで，大きく変換時間を短縮することが可能である．図 5.17 に示す PatComp(8) は，Algorithm6 中の波線部の処理を 8Thread で並列して行った場合の変換時間である．図から確認できる通り，並列化を行うことで PatComp への変換時間が大幅に短縮されている．Queen_4147 では 791 倍高速に変換を行うことが出来ている．ある Thread が $Table$ から一致するパターン

を発見し次第，パターンを確認する処理は打ち切られるため，8 倍以上の高速化に成功した．また今回使用した CPU は Intel Core i7 6700K であり，最新の CPU に比べ 4 世代前の CPU である．そのため最新の CPU を用いて並列度を上げることでさらに大幅な高速化が可能であり，他の疎行列格納方式の変換と同等以上の速度で変換が行えると考えている．

本研究で提案した PatComp 法の COO 形式からの変換時間は既存疎行列格納方式に比べ，長い結果となった．しかしながら FEM のモデルが複雑になるにつれ反復法の反復回数は大幅に増え，疎行列格納方式への変換は 1 回しか行われないことから，全体の反復法処理時間中，変換時間が占める割合は小さくなる．

また FDM や FEM で使用するモデルの形が変わらない限り，疎行列内の非ゼロ要素の位置が変わらない．よって一度 PatComp 法へ変換を行えば，パラメータ等を変え再度シミュレーションを実行する際に，PatComp 法の非ゼロ要素の位置を示す配列を再利用することが可能である．

4.9 PatComp の適用可能な疎行列

PatComp は COO からの変換時間が他の疎行列格納方式に比べ長くなっていることから，数値シミュレーション中に多くの PatComp への変換が必要となる場合，PatComp の適用が困難となる．通常，FEM を用いた数値シミュレーションでは各時間ステップごとに反復法を用いて連立一次方程式を解くことになる．そのため各時間ステップごとに PatComp で格納された疎行列を用意する必要があるが，COO から PatComp への変換コストは非常に大きいため，毎回 COO から PatComp への変換を行っていると変換時間がシミュレーションのボトルネックとなる．しかしながら疎行列内の非ゼロ要素の位置に関しては FEM モデルの形状が変化しない限り，シミュレーション終了まで変わることがない．ここで FEM モデルの形状が変化しないとは，シミュレーションの解析対象に破壊や亀裂等が発生せず，FEM モデル中のメッシュの構成にも変化がないことを意味する．一般的にメッシュの構造と疎行列の位置には関連があることから，メッシュ構造に変化がなければ疎行列の非ゼロ要素の位置も変化することがない．このように疎行列中の非ゼロ要素の位置が変化しない場合，PatComp の非ゼロ要素の位置に関わる配列 $Offset$，$Table$，$TableIndex$，$ValPtr$ の内容を時間ステップごとに更新する必要がなく，$PatCompVal$ の内容のみが更新される．よって数値シミュレーションの最初の時間ステップにて PatComp への変換に長時間必要とするが，その後の時間ステップにおいて PatComp への変換コストは非常に小さくなる．解析が難しい数値シミュレーションでは時間ステップ数が非常に多くなり，相対的に PatComp への変換時間が全体の処理時間を占める割合は小さくなり，SpMV 等の反復法の処理が支配的となる．よって数値シミュレーション中に FEM モデルの形状が変化しない問題に対して PatComp は有効であり，メモリ使用量を大きく削減し数値シミュレーションの規模，精度を向上させることが可能である．一方で，数値シミュレーションの解析対象が時間ステップごとに破壊，亀裂等による変形する場合，FEM モデル中のメッシュの構造も変化する．この場合には疎行列内の非ゼロ要素の位置もメッ

シュの構造の変化に伴い変化する．よってPatCompを用いる場合，各時間ステップごとに$Table$等の非ゼロ要素の位置に関わる配列の更新も必要となり，シミュレーション終了まで多くの変換コストが必要となる．このような問題に対してはPatCompを適用することは困難だと考えられる．

またメモリ使用量の評価実験において，PatCompを使用した場合CSRに比べメモリ使用量が増加する疎行列も存在した．そこでPatCompが有効な疎行列を判別できるよう，既存疎行列格納方式であるCSRのメモリ使用量に比べPatCompのメモリ使用量が少なくなる条件式を導出する．式（4.2）にCSRのメモリ使用量に比べPatCompのメモリ使用量が少なくなる条件式を示す．式に使用する変数は疎行列の行数Nと疎行列内に存在するパターン数$N_{pattern}$である．

$$\begin{aligned} 8N_z + 4N_z + 4(N+1) &> 8N_z + 4(N+1) + 4N + 8N + 4N_T \\ &\approx 8N_z + 4(N+1) + 4N + 8N + 4(N_z/N \times N_{pattern}) \end{aligned} \tag{4.2}$$

式（4.2）の左辺はCSRのメモリ使用量を導出する式を示しており，右辺がPatCompのメモリ使用量を導出する式を示している．節4.7.1では，PatCompの$Table$のサイズN_Tを用いてPatCompのメモリ使用量を算出していたが，算出式を簡易化するためにN_Tを$N_z/N \times N_{pattern}$で近似することでメモリ使用量を算出する．N_z/Nは疎行列の1行あたりの平均要素数を表す．疎行列内の総パターン数を調べることができれば，おおよそのPatCompのメモリ使用量を導出可能となる．また疎行列からでなく，FEMモデルを構成する際に，同じ繋がり方の節点が何個あるかという情報を調べておくことで，その問題がPatCompに適しているか大まかに判断することが可能である．

次にメモリ使用量の評価実験の結果から，PatCompのメモリ使用量が少なくなる疎行列の特徴について，定性的に評価する．評価実験の結果からPatCompのメモリ使用量が少ない疎行列の特徴は以下の通りである．

- 疎行列内の行あたりの平均要素数(N_z/N)が多い
  1行あたりの非ゼロ要素数が多い場合，パターンが重複した際の削減率が高くなり，また連続した非ゼロ要素が存在する確率も高くなる．連続した非ゼロ要素が多くなると連続した列番号を1つの連続数の値で表すことができ，メモリ使用量削減の効果が大きくなる．

- 疎行列内のパターン数が少ない
  疎行列中の全行を少ないパターンで表せるほど非ゼロ要素の位置情報格納のためのメモリ使用量が少なくなる（$Table$のサイズが小さくなる）．

- 疎行列内の連続した非ゼロ要素が多く存在する
  連続した非ゼロ要素の列番号はCSR等の既存手法だと全ての非ゼロ要素の列番号を格納している．PatCompでは連続した場合，非ゼロ要素の連続数で格納するため，

どれだけ長く連続した列番号も 1 つの要素のみで表すことができる．よって連続した非ゼロ要素が多く存在するほどメモリ使用量が小さくなる．

- 疎行列内に不連続な非ゼロ要素が少ない
  PatComp において不連続な非ゼロ要素の位置情報を格納するために，1 という連続数とその後の 0 の個数の 2 つの情報が必要となる．そのため，CSR に比べても 1 つの非ゼロ要素の列番号を表すために必要となるメモリ使用量が多くなる．よって不連続な非ゼロ要素が少ないほど無駄のない格納方式となる．

4.10 おわりに

本章では既存疎行列格納方式よりさらにメモリ使用量削減するため，パターン性を考慮した Pattern Compression (PatComp) 法を提案した．既存手法で疎行列を小さく分割した部分行列という小さい範囲での圧縮を採用しているが，さらに広い範囲を少ないデータ量で表すことで，メモリ使用量の削減率を向上させることが可能である．PatComp 法は FDM や FEM で生成させる疎行列には，1 行という広い範囲にパターン性が存在することに着目した手法である．各行のパターンを使用することで変換時間は比較的長くなる可能性が大きいが，時間発展に伴い疎行列の形状が変化しない FEM モデルでは変換時間が支配的にならないことから，PatComp 法を用いることで従来と演算性能は変わらず，メモリ使用量を大幅に削減することが可能である．

PatComp の有効性を確認するため，4.2 節において，実際に FEM で生成された非ゼロ要素にパターン性が存在するか検証を行った．その結果多くの疎行列においてパターンが存在することを確認し，複数の行が同一のパターンとなることを明らかにした．この複数の行の共通のパターンを格納しておくことで，メモリ使用量の大幅な削減が可能である．

4.2 節の結果を踏まえ，4.4 節において PatComp 法を提案した．PatComp 法は，疎行列中の全てのパターンを $Table$ 行列に格納し，各行はどのパターンに属するかを示す目印を保持するだけで，非ゼロ要素の列番号を得ることができる．従来の格納方式では各行全ての非ゼロ要素の列番号を保持していたが，疎行列内の各行で共通するパターンを格納しておくことで，複数の行の非ゼロ要素の列番号を少ないメモリ使用量で表現することが可能となる．

また 4.5 節にて PatComp を用いた SpMV カーネルを提案した．PatComp では $Table$ から各 Thread が担当する行のパターンを読み出し，そのパターンに則り SpMV の演算を行う．

本章において行った評価実験ではメモリ使用量，SpMV 演算性能，GMRES を用いた演算時間を既存疎行列格納方式である CSR，ELL，ELL-R と PatComp 法を比較した．評価実験のため，実際の数値シミュレーションで使用された疎行列を Florida sparse matrix collection を使用した．

メモリ使用量の評価では，多くの疎行列において PatComp 法を使用し，メモリ使用量の削減を達成した．

PatComp は疎行列内の非ゼロ要素の連続性だけでなく，非ゼロ要素の並び，パターン性を考慮した手法である．そのため疎行列から PatComp への変換時間は他の疎行列格納方式に比べ，多く必要となるが，高いメモリ使用量の削減が期待できる．結果として，CSR と比較すると平均 19.4%，cant と consph において最大 31.1%のメモリ使用量削減に成功した．また 15 個中 4 個の疎行列において 30%以上，12 個の疎行列において 25%以上のメモリ使用量削減を達成した．疎行列内の非ゼロ要素の連続性のみを考慮した RBP-CSR に比べ，平均で 4.3%，UT_Heart2 において最大 14.1%のメモリ使用量を削減した．

続く SpMV 演算時間の評価では既存疎行列方式 CSR と比較して，PatComp 法の SpMV 演算性能が遜色ないことを確認した．

最後の GMRES の演算時間の評価では，PatComp 法を GMRES に適用した際の変換時間を含めた演算時間を評価した．PatComp の変換は他の疎行列格納方式に比べ並列化を行わない場合長い時間を要した．PatComp は元々，時間ステップごとに疎行列の形が変わらない問題に適用することを前提に提案した手法である．そのため疎行列から PatComp への変換は，数値シミュレーションの最初に 1 回実行するだけで，その後のシミュレーションでは PatComp の位置情報を使い回すことができる．そのため変換時間が多少多くかかっても PatComp を用いてメモリ使用量を削減する有用性は高いと考えられる．

以上の結果から，PatComp 法を使用することで，既存疎行列格納方式の演算性能はそのままに，メモリ使用量を最大で 30%以上削減できることを確認した．PatComp 法を用いてメモリ使用量を削減することで，GPU を使用したオンサイト環境においても，さらに大規模かつ高精度な数値シミュレーションが可能となった．オンサイト環境で高精度なシミュレーションを行うことで，様々な分野に貢献できると考えている．

第5章　数値シミュレーションの時間発展に伴いFEMモデルの形状が変化する問題に対する疎行列格納方式：Row Block Packing method (RBP法)

5.1　はじめに

4章では疎行列中の各行のパターンを考慮した疎行列格納方式PatCompを提案した．PatCompは広い範囲でのパターンを用いてデータ量の削減を行うためメモリ使用量を大幅に削減することが可能である．しかしながら変換には非常に多くの時間を要するため，時間ステップごとにFEMモデルの格子形状が変化せず，疎行列の形が変わらない問題にのみ有効であった．そこで本章で提案するRow Block Packing（RBP）法は疎行列からの変換が高速かつ既存の疎行列格納方式に比べメモリ使用量が少ない手法を目的とする．そのためにPatCompでは行単位のパターン性を用いてメモリ使用量の削減を行っていたが，RBP法では行中の連続した非ゼロ要素に着目する．圧縮を行う単位を連続した非ゼロ要素という小さな範囲に限定することで，PatCompのように各行のパターンをTableに登録されているか参照する手間が必要なく，非ゼロ要素の位置情報を全て1回探索するだけで圧縮を行うことが可能である．BCCOO等の既存手法においても小さい範囲での圧縮を行っていたが，演算性能の向上やメモリ使用量削減の最適化のため，長い時間をチューニングに費やしていた．そのため変換時間はRBP法より長くなる．

RBP法は連続した非ゼロ要素の先頭と末尾の列番号のみを格納することで，列番号に関する情報量を減らし，メモリ使用量の削減を行う．また不連続な非ゼロ要素に関しては既存手法であるCSRで格納することで，無駄な要素の追加とメモリアクセス効率の悪化を抑制する．

5.2節にて，RBP法の原理について述べ，5.3節ではRBP法の具体的な変換手順について説明する．続いて5.4節，5.5節にて，実際にCSRに対してRBP法を適用したRBP-CSR

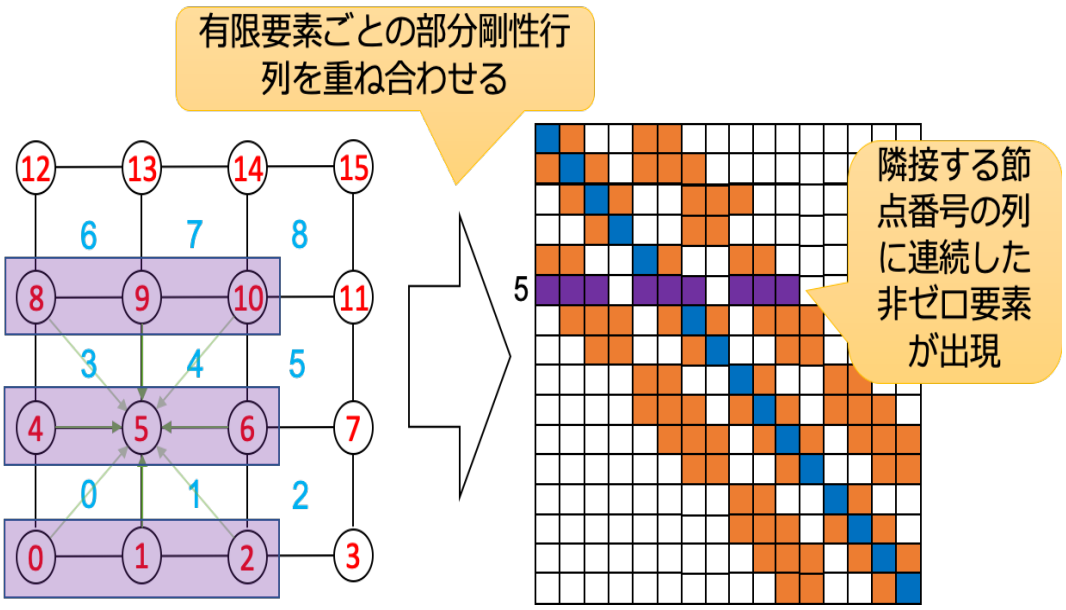

図 5.1: FEM で生成される疎行列内の非ゼロ要素の連続性

と ELL に適用した RBP-ELL への変換方法と SpMV カーネルについて述べる．

その後 5.6 節から RBP 法の評価実験を行う．提案した疎行列格納方式 RBP 法（RBP-CSR，RBP-ELL）の評価を行う．RBP 法は既存手法である CSR，ELL の SpMV 演算性能を保ちながらメモリ使用量を削減することを目的としている．また時間発展に伴い疎行列の形状が変化する問題や反復法の収束が早い問題では，変換時間が重要となることから，RBP 法は変換の手順を簡潔にし，高速に変換できるよう設計されている．本評価実験ではこれらの RBP 法の趣意が達成されているか確認する．

まず実際のシミュレーションで用いられた疎行列を用いて，各疎行列格納方式のメモリ使用量を測定する．次に，GPU 上における各疎行列格納方式を用いた SpMV の演算性能を評価する．一般的に，メモリ使用量を削減することで SpMV 演算性能は低下する傾向があることから，RBP 法を使用した場合の演算性能に対する影響を検証する．圧縮を施した場合でも既存の疎行列格納方式の SpMV 演算性能と遜色がないことが理想である．最後に非定常反復法の 1 例として GMRES を使用した評価を行う．RBP 法への変換時間を含めた GMRES の演算時間を評価し，変換時間が反復法のボトルネックとなっていないことを検証する．PatComp を用いた GMRES にてボトルネックとなっていた疎行列からの変換時間が短縮されていることを確認する．

5.2 RBP 法の原理

FEM で扱う疎行列の生成例を図 5.1 に示す．生成される疎行列中のある 1 行において，ある一つの節点とその周りの節点の番号に対応する疎行列中の列位置に非ゼロ要素が現れ

る．例えば図 5.1 の例では，節点 5 は節点 0，1，2，4，6，8，9，10 に囲まれている．この場合，疎行列中の 5 行目（節点番号に対応）には，0，1，2，4，5，6，7，8，9，10 に非ゼロ要素が出現する．そのため “0，1，2”，“4，5，6”，“7，8，9” のように連続した非ゼロ要素が存在する．RBP 法ではこの連続性を利用し，メモリ使用量の削減を行う．FEM では，ある節点とその隣接した節点の情報から方程式を作り，計算を行うことから，節点は必ず 1 個以上の節点と隣接することとなる．そのため多くの場合疎行列内には，連続した非ゼロ要素が存在する．

そして FEM で生成した疎行列を使用する処理 SpMV は，疎行列内に非ゼロ要素が連続して存在する場合，連続した非ゼロ要素の最初と最後の要素の列番号のみで計算可能である．

既存の格納方式である CSR や ELL では，連続した非ゼロ要素の全ての列番号を記憶しているが，Row Block Packing (RBP) 法では，疎行列内に存在する連続した非ゼロ要素の列番号の最初と最後のみを格納することで，列番号を格納する配列に要する記憶領域を削減することが可能である．従来の CSR，ELL 等の疎行列格納方式では連続性を考慮した格納がされておらず，多数の疎行列格納方式のメモリ使用量を削減することが可能であると考える．本研究では代表的な疎行列格納方式である CSR と ELL に対し，連続した非ゼロ要素を RBP 法により圧縮することでメモリ使用量を削減する．RBP 法の考え方を用いることで連続した非ゼロ要素に圧縮を行っていない Sliced ELL や SELL-C- σ 等のメモリ使用量を削減することも可能である．

PatComp においても連続した非ゼロ要素の連続性を考慮して圧縮を行っていたが，連続した非ゼロ要素の先頭位置のみを格納し，そこから何個非ゼロ要素が存在するか確認するためには Table 内に登録されているパターンを使用する必要があった．PatComp ではテーブルを作成するため多くの時間を費やしていたため，連続性を用いても変換時間は長い．RBP 法では PatComp のような 1 行のパターンを利用して連続した非ゼロ要素を復元するのではなく，単純に連続した非ゼロ要素の先頭と末尾の列番号のみを格納する．この方式を取ることで圧縮の際，テーブルの参照等が必要なく，連続した非ゼロ要素の先頭と末尾を順に確認していくだけで良い．よって PatComp に比べ非常に高速な変換が可能となる．

RBP 法のように非ゼロ要素の連続性を用いた疎行列格納方式として BCCOO や ABM の疎行列格納方式が存在する．この二つの疎行列格納方式において連続した非ゼロ要素の列番号の連続性を考慮したメモリ使用量の削減が行われている．CoAdELL[39]，BCOOO[41] や ABM[42] の研究目的は格納する列番号数の減らすことによるメモリアクセス回数の削減と連続した GPU メモリへのアクセスの実現による SpMV 演算性能の向上である．演算性能の向上のためチューニングを必要とすることから，疎行列を変換する際，ベンチマークとして仮に複数 SpMV を実行したり，パラメータを決定するため複雑な計算を行う必要がある．そのため各疎行列格納方式への変換には PatComp と同様に多くの時間を必要とする．

解析対象の FEM モデルが亀裂や破壊により形状変化が起こる数値シミュレーションで

Algorithm 8 RBP 法の適用フロー

```
1: for i = 0 to 疎行列内の非ゼロ要素数 do
2:   if i 番目の非ゼロ要素が連続 then
3:     非ゼロ要素の値を対応する BLOCK の Values へ格納
4:     if i 番目の非ゼロ要素が連続した要素の先頭 or 末尾 then
5:       非ゼロ要素の列番号を対応する BLOCK の Columns へ格納
6:     end if
7:     BLOCK を適用する疎行列格納方式で格納
8:   else if i 番目の非ゼロ要素が不連続 then
9:     不連続部の要素はすべて CSR 形式で格納
10:  end if
11: end for
```

は，時間発展に伴い格子の形状が変化する．格子の変化は疎行列の形状にも影響するため時間ステップごとに各疎行列格納方式への変換が必要となる．そのため時間発展に伴い格子が変化するシミュレーションでは，疎行列格納方式の変換時間がボトルネックとなってしまうため，高速に変換可能かつメモリ使用量の削減が可能な手法が必要とされる．また少数の反復回数で反復法が収束する問題においても変換時間が支配的になり，高速化のボトルネックとなる．

RBP 法は複雑なチューニングやベンチマークの実行を必要とせず疎行列内の非ゼロ要素を 1 回ずつ参照することで変換可能であることから，PatComp，BCCOO や AMB と比べ変換時間が少なく，時間発展ごとに格子形状が変化する問題や反復法が早期に収束する問題において有効であると考えられる．RBP 法は，疎行列格納のために必要なメモリ使用量を削減しつつ高速に変換可能な手法を目的としている．

RBP 法によりメモリ使用量を削減することで，少量の GPU メモリを効率的に使用することで，オンサイト環境におけるシミュレーションの高精度化，大規模化，同時に実行できるシミュレーション数の増加につながる．RBP 法は FEM で生成される疎行列を対象としており，連続な非ゼロ要素がほぼ存在しないような疎行列に対しては提案手法の効果は薄い．

5.3 RBP 法

図 5.2 に RBP 法の概略図，Algorithm8 に RBP 法を既存の疎行列格納方式に適用する手順を示す．

Algorithm8 では，疎行列内の非ゼロ要素を先頭から順に，図 5.2 中，斜線の四角で囲まれた要素のように連続しているか，もしくは図 5.2 中，丸で囲んだ要素のように不連続なのかを確認していく．RBP 法では，圧縮を行う単位を 1 行中の連続した非ゼロ要素に限定することで，非ゼロ要素の列番号に必要となる要素の参照回数，計算量が少なくなる．

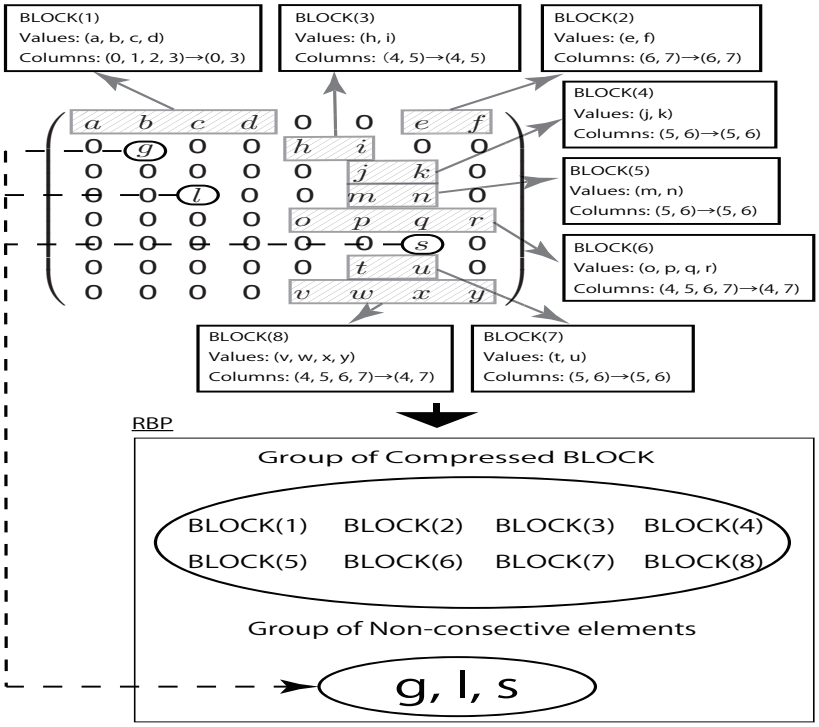

図 5.2: RBP 法の概略図

疎行列内の非ゼロ要素を 1 つずつ確認していくだけで，非ゼロ要素が連続しているかどうかを確認でき，連続した非ゼロ要素の場合には列番号を削減することでメモリ使用量を削減することが可能である．

BCCOO では疎行列を複数行または複数列に渡る小さいブロックに分割し，その基準となる位置情報のみを格納することで，位置情報の削減を行う．RBP 法は変換の速度を重視するため各行中の連続した非ゼロ要素を対象としているため，BCCOO に比べメモリ使用量の削減率が低くなる可能性がある．しかし BCCOO では範囲の広いブロックを用いることで，SpMV に不必要な 0 要素が含まれメモリ使用量が増加する可能性があることから，メモリ使用量が最も少なくなるよう最適化するため，複数のブロックサイズでの変換を試し，最も良い結果を採用する．そのため疎行列内の非ゼロ要素に対する参照回数は RBP 法の 1 要素 1 回に比べ数倍となる．よって RBP 法に比べ BCCOO は変換時間が大幅に長くなる．

まず Algorithm8 中，2 行目から 7 行目までは，非ゼロ要素の連続部に対して行う処理である．RBP 法では図 5.2 中 “a, b, c, d” 等のように，連続した非ゼロ要素の情報に関して 1 ブロックとして考え，最終的に RBP 法を適用する疎行列格納方式へブロック単位で格納する．また，連続した非ゼロ要素の値については圧縮の影響がないため，連続している間，Values にただ格納するだけである (8，3 行目)．

4 行目から 6 行目に示す処理は非ゼロ要素の列番号数を削減する処理であり，RBP 法の最も重要な処理である．各ブロックの $Columns$ のデータ数を削減するため，SpMV の計算に必要な先頭と末尾の要素のみを取り出し格納する（図 5.2 中，各ブロックの $Columns$ の矢印左側がデータ数を削除する前，矢印右側がデータを削除した後の $Columns$ の配列である）．この結果，どれだけ長く連続した非ゼロ要素を BLOCK 化した場合でも，列番号を表すために必要となる要素数は 2 となるため，疎行列内に連続した非ゼロ要素が多く存在したり連続数が長い場合，大きなメモリ使用量の削減が可能である．

7 行目の「BLOCK を適用する疎行列格納方式で格納」は，既存の疎行列格納方式に対して，ブロック化した非ゼロ要素を格納していく処理である．格納する際には，各ブロックを本来の 1 つ非ゼロ要素と同等に扱い格納する（つまり $Values$ と $Columns$ の複数の要素を一つの塊として考える）．この過程は，CSR や ELL などの既存の格納方式にて値や列番号などを格納する方法をブロック化された非ゼロ要素に対して適用することを意味する．例えば CSR であれば，ブロック化された非ゼロ要素の複数の値をそのまま値配列に，圧縮された 2 つの列番号は列番号の配列に，1 つの値を扱う場合と同様に格納する．

8 行目，9 行目では，i 番目の非ゼロ要素が不連続だった場合の処理を記述している．図中の “g, l, s” のような不連続な非ゼロ要素を CSR 形式で格納する．この時，ELL などの CSR とは異なる格納方式においても，不連続な非ゼロ要素は連続した非ゼロ要素とは別の空間に CSR 形式で格納する．連続部と不連続部を分離して格納する理由は，メモリアクセス効率を向上させるためである．RBP 法のように，不連続な非ゼロ要素については CSR 方式で格納することで，連続した非ゼロ要素の列番号を格納している配列または行列は必ず連続した非ゼロ要素に対しアクセスすることとなる．これによりスレッドが 1 回の反復で必ず 2 つの値を参照する．1 スレッドのアクセス数が固定であれば，warp 内のスレッドが一定のインデックスでスライドし，アクセスすることが可能である．そのため，warp 内の各スレッドは常にコアレスドアクセスを行うことが可能である．

ただし圧縮により，既存格納方式と同様の処理では格納できない場合も存在する．その際には配列の追加等，正しく SpMV の計算が行えるように変更を加える必要がある．本稿では代表的な疎行列格納方式である CSR，ELL の格納方式に則ったブロックの格納方法を，図 5.3 を用いて次の節から説明を行う．

RBP 法は複数の値をブロック化するというシンプルな方法を取っているため，他の疎行列格納方式に対しても，多少の変更のみで同様に RBP 法を適用可能であると考えられる．例えば，ELL の派生系である ELL-R は ELL と同様の方法で RBP を適用可能である．

PatComp では，COO 形式で格納されている FEM モデルで生成された疎行列を PatComp へ直接変換し，反復法の SpMV で使用することを想定している．RBP-CSR では CSR で格納されている疎行列からの変換，RBP-ELL では ELL で格納されている疎行列からの変換を行い，反復法内の SpMV で使用することを想定する．既存疎行列格納方式では GPU メモリ容量の制約で数値シミュレーションが実行できない場合，容易に適用可能なメモリ使用量削減手法が RBP 法である．現在多くの数値シミュレーションにおいて CSR や ELL で疎行列が格納されているため，利便性を高めることを目的に本稿では CSR

から RBP-CSR へ，ELL から RBP-ELL への変換を想定する．ただし COO から直接変換を行うことも容易に実装することが可能である．

5.4 CSR への RBP 法の適用 (RBP-CSR)

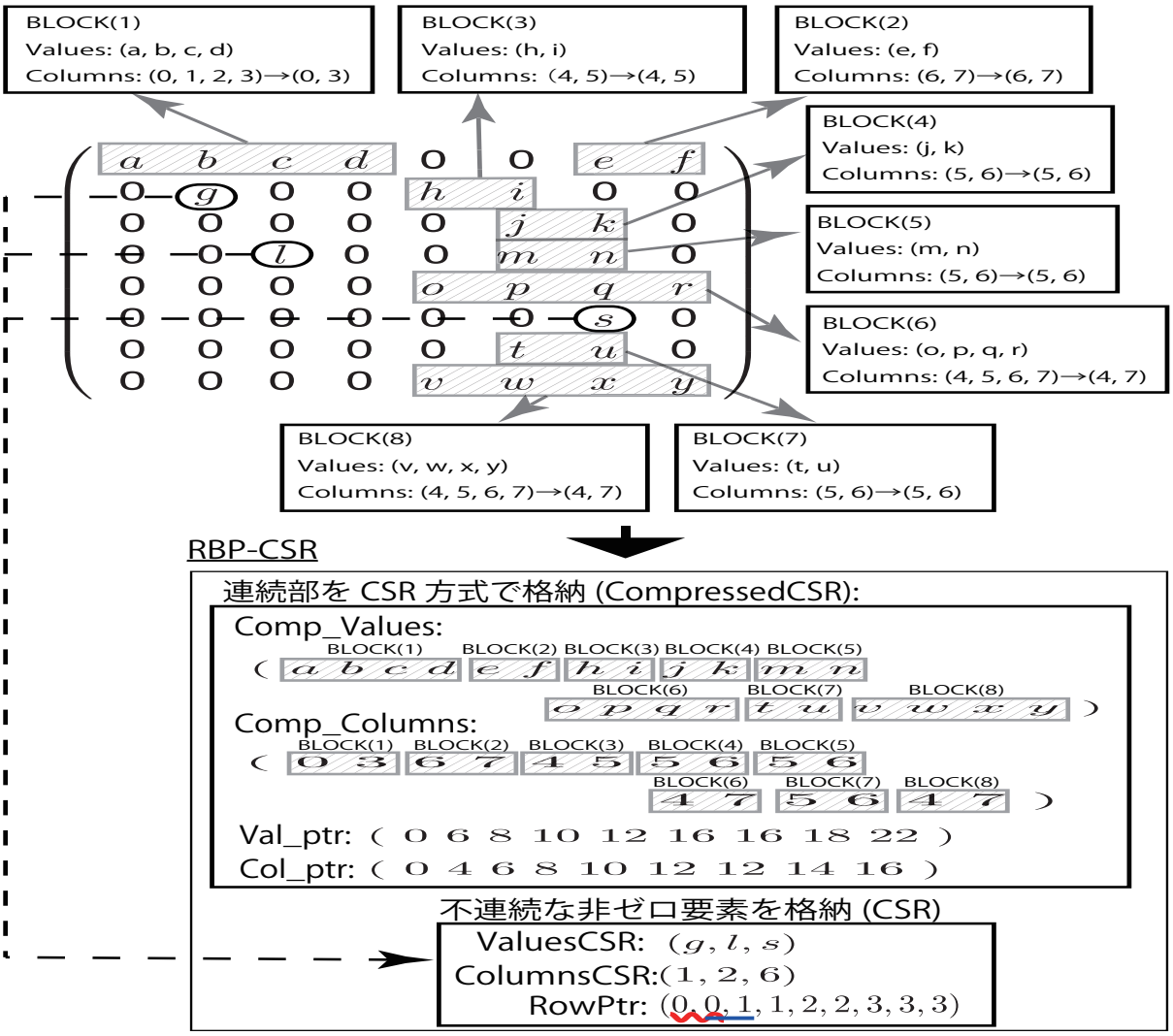

図 5.3: RBP-CSR

図 5.3 に RBP 法を施した CSR(RBP-CSR) の概要を示す．RBP-CSR は Algorithm9 に則り，CSR の列番号を格納する配列の要素数を削減していく．Algorithm9 は非ゼロ要素が 2 つ以上連続しているか判断するために，Algorithm8 に比べ複雑に見えるが，同じ流れで処理を行っている．Algorithm9 の 3 行目と 5 行目は元の CSR から疎行列内の非ゼロ要素の情報を 1 つずつ取り出すためのループ処理である．$RowPtr[i]$ から $RowPtr[i]-1$

までが i 行目の非ゼロ要素の情報が格納されている，$Values$，$Columns$ 配列のインデックスとなる．Algorithm9 中の $Col_{CSR}, Val_{CSR}, RowPtr_{CSR}$ は図 3.8 の CSR の配列に対応している．また，その他の配列は図 5.3 に対応している．

Algorithm9 の 6 行目から 23 行目までが Algorithm8 の 2 行目から 7 行目までの非ゼロ要素の連続部の処理に対応する．6 行目からの While 処理により $TmpCol$ に格納された一つ前の非ゼロ要素の列番号と，その次の非ゼロ要素の列番号が連続している間，$Comp\_Values$ へ非ゼロ要素の値を追加し続ける (Algorithm8 の 3 行目に対応)．Wlile 処理の終了時には，$Comp\_Columns$ に 2 つ以上連続した非ゼロ要素の先頭と末尾の列番号が追加される (Algorithm8 の 4 から 6 行目に対応)．例えば，図 5.3 の疎行列の最初の “a, b, c, d” は連続した非ゼロ要素であることから，連続した非ゼロ要素の値はすべてそのまま $Comp\_Values$ へ格納される．また，先頭の非ゼロ要素 “a”，末尾の要素 “d” の列番号がそれぞれ順に $Comp\_Columns$ に格納される．

Algorithm8 の 8 から 10 行目までの不連続部の要素に対する処理は，Algorithm9 の 24 行目から 29 行目に対応する．$TmpCol$ に各格納されている列番号の非ゼロ要素が，不連続と判明した場合に，非ゼロ要素の値を $ValuesCSR$ へ，列番号を $ColumnsCSR$ へ追加する．例として，図 5.3 の疎行列中，最初の不連続な非ゼロ要素 “g” の非ゼロ要素は，8 から 10 行目の処理が適用され，“g” の値と列番号はそれぞれ，$ValuesCSR$ と $ColumnsCSR$ へ格納される．

Algorithm9 の 30 行目から 40 行目は，調査している非ゼロ要素が 1 行中の最後の非ゼロ要素が不連続だった場合の処理と，1 行中に 1 つの不連続な非ゼロ要素しか存在しない場合の処理を示している．この 2 つの処理は実装の都合上，追加したものであり，Algorithm8 の不連続部の要素の処理に含まれる．

また，RBP 法を CSR へ適用することで多少の変更が生ずる．図 5.3 の疎行列を CSR で格納する場合，値と列番号は 1 対 1 で対応することから，1 行目の要素は $Values$ が “a, b, c, d”, “e, f”, $Columns$ が “$0, 1, 2, 3$”, “$6, 7$” のようにそれぞれの配列で要素数が一致し，Ptr 配列は $RowPtr$ の 1 つのみ使用する．しかし RBP 法の圧縮により，値と列番号の数に差異が生じるため，RBP-CSR では 2 行目の要素が始まるインデックスは $Comp\_values$ で 6 番目，$Comp\_Columns$ では 4 番目となる．そのため RBP 法では $Comp\_Values$, $Comp\_Columns$ の何番目のインデックスで行が変わるのかを表す値を格納する Val_ptr，Col_ptr の 2 つの配列を用意し，並列計算に必要な各行の先頭の要素のインデックスを判別する．Algorithm9 の 41 行目にて，不連続部を格納する CSR の配列，Comp_Values 配列，Comp_Columns 配列，それぞれに対して，各行の先頭の非ゼロ要素が配列内の何番目のインデックスに格納されているか記録するため，各行終了時点での非ゼロ要素数を格納している．RBP-CSR では追加の配列を必要とするため，全く連続した非ゼロ要素が存在しない疎行列においては CSR よりメモリ使用量が増える可能性がある．

さらに，連続部と不連続部の分割により，5 行目（最初の行を 0 行目とする）のような連続した非ゼロ要素が存在しない行も出現する．その場合には Val_ptr, Col_ptr に同じインデックスを続けて格納することで，行中に要素が存在しないことを示す．図 5.3 の行列

では 1 行目には不連続な非ゼロ要素が存在せず，2 行目から不連続な非ゼロ要素 “g” が出現する．そのため，$RowPtr$ の 1 番目と 2 番目に “$0, 0$” (図 5.3 中，赤い波線) を格納する．2 行目は $RowPtr$ 内 2 番目と 3 番目 “$0, 1$” (図 5.3 中，青い直線) のように非ゼロ要素 “g” が 1 つ存在することを示す．上記に説明した Algorithm9 を図 5.3 の疎行列中，すべての非ゼロ要素に対し，適用することで図 5.3 中の RBP-CSR を得ることが可能である．

式 (5.1) に RBP-CSR を用いて疎行列を格納した場合のメモリ使用量を示す．以降，N_{col} は圧縮した後の列番号を格納する $Comp\_Columns$ の要素数，N_{val} は $Comp\_Values$ の要素数，N_{non} は $ValuesCSR, ColumnsCSR$ の要素数 (疎行列内の不連続な非ゼロ要素の数) を示す．

図 5.3 中の Val_ptr, Col_ptr, $RowPtr$ の要素数は，CSR の $RowPtr$ と同様に $(N+1)$ となる．そのため，式 (5.1) 中 $12(N+1)$ は，3 つの配列 (Val_ptr, Col_ptr, $RowPtr$) のメモリ使用量の合計を示す．次に $4N_{\mathrm{col}} + 8N_{\mathrm{val}}$ は連続した非ゼロ要素の列番号と値を格納する $Comp\_Columns$ と $Comp\_Values$ のメモリ使用量を示す．$4N_{\mathrm{non}} + 8N_{\mathrm{non}}$ は不連続な非ゼロ要素の列番号と値を格納する $ColumnsCSR$，$ValuesCSR$ のメモリ使用量を示す．

$$\begin{aligned} MemUsage_{RBP-CSR} = 12(N+1) + 4N_{\mathrm{col}} + \\ 8N_{\mathrm{val}} + 4N_{\mathrm{non}} + 8N_{\mathrm{non}}\ [byte] \end{aligned} \tag{5.1}$$

Algorithm 9 Matrix compression of RBP-CSR

1: $Val\_ptr[0] = 0, Col\_ptr[0] = 0, RowPtr[0] = 0$
2: $Cnt1 = 0, Cnt2 = 0, Cnt3 = 0$
3: **for** $i = 0$ to $NumRow$ **do**
4: $TmpCol = Columns_{CSR}[RowPtr_{CSR}[i]]$
5: **for** $j = RowPtr_{CSR}[i] + 1$ to $RowPtr_{CSR}[i + 1]$ **do**
6: **while** $Columns_{CSR}[j] - TmpCol == 1$ **do**
7: **if** $TmpCol$ is head of consecutive columns **then**
8: $Comp\_Columns[Cnt1] = TmpCol$
9: $Comp\_Values[Cnt2] = Values_{CSR}[j - 1]$
10: $Comp\_Columns[Cnt1 + 1] = Columns_{CSR}[j]$
11: $Comp\_Values[Cnt2 + 1] = Values_{CSR}[j]$
12: $Cnt2 = Cnt2 + 2$
13: **else**
14: $Comp\_Columns[Cnt1 + 1] = Columns_{CSR}[j]$
15: $Comp\_Values[Cnt2] = Values_{CSR}[j]$
16: $Cnt2 = Cnt2 + 1$
17: **end if**
18: $TmpCol = TmpCol + 1$
19: $j = j + 1$
20: **end while**
21: **if** Previous elements is consecutive **then**
22: $TmpCol = Columns_{CSR}[j]$
23: $Cnt1 = Cnt1 + 2$
24: **else**
25: $ColumnsCSR[Cnt3] = TmpCol$
26: $ValuesCSR[Cnt3] = Values_{CSR}[j - 1]$
27: $Cnt3 = Cnt3 + 1$
28: $TmpCol = Columns_{CSR}[j]$
29: **end if**
30: **if** $TmpCol$ is last element's column in the row **then**
31: $ColumnsCSR[Cnt3] = TmpCol$
32: $ValuesCSR[Cnt3] = Values_{CSR}[j]$
33: $Cnt3 = Cnt3 + 1$
34: **end if**
35: **end for**
36: **if** There is just one element in the row **then**
37: $ColumnsCSR[Cnt3] = TmpCol$
38: $ValuesCSR[Cnt3] = Values_{CSR}[RowPtr_{CSR}[i]]$
39: $Cnt3 = Cnt3 + 1$
40: **end if**
41: $Col\_ptr[i + 1] = Cnt1, Val\_ptr[i + 1] = Cnt2, RowPtr[i + 1] = Cnt3$
42: **end for**

Algorithm 10 SpMV code of RBP-CSR on GPU

1: Let id be thread ID (id: 0 to n-1)
2: Set $Count, TmpReslut = 0$
/* Calculation of compression part */
3: $RowStart = col\_ptr[id]$
4: **for** $jj = val\_ptr[id]$ to $val\_ptr[id+1] - 1$ **do**
5: $ColNow = Comp\_Columns[RowStart + Count]$
6: $ColNext = Comp\_Columns[RowStart + Count + 1]$
7: $TmpResult+ = Comp\_Values[jj] * dVector[ColNow]$
8: **for** $j = ColNow + 1$ to $ColNext$ **do**
9: $jj++$
10: $TmpResult+ = Comp\_Values[jj] * dVector[j]$
11: **end for**
12: $Count+ = 2$
13: **end for**
/* Calculation of non-compression part */
14: $RowStart = RowPtr[id]$
15: $RowEnd = RowPtr[id+1]$
16: **for** $j = RowStart$ to $RowEnd - 1$ **do**
17: $TmpResult+ = Values[j] * dVector[Colmuns[j]]$
18: **end for**
19: $Result[id] = TmpResult$

Algorithm10 は，RBP-CSR 形式で格納された疎行列を用いた SpMV 演算を示す．各スレッドが Algorithm10 を同時に実行し，それぞれが疎行列内の 1 行を担当する．id はカーネル関数を実行する各スレッドの番号を示している．id は，CUDA の関数を用いることで各スレッドが取得することが可能である．RBP-CSR の SpMV カーネルは連続した非ゼロ要素を計算するパートと不連続な非ゼロ要素を計算するパートに分かれている．Algorithm10 内，3 行目から 13 行目までが連続した非ゼロ要素を計算するパートである．4 行目からの for ループでは，スレッドが担当する行の非ゼロ要素の値を $Comp\_Values$ から値を読み出すため，Val_ptr から担当する行の先頭の非ゼロ要素の値が格納されているインデックス番号を読み込む．5,6 行目の $ColNow$ と $ColNext$ には連続した非ゼロ要素の最初と最後の列番号が代入される．7 行目の処理は，連続した非ゼロ要素の先頭の要素と SpMV で用いられるベクトルの要素との掛け算を計算する．8 行目の for ループにて，$ColNow$ から $ColNext$ になるまで j をインクリメントしていくことで，先頭以降の要素の列番号を毎回グローバルメモリから読み出すことなく，インクリメントのみで列番号を復元し，SpMV の演算を行う．レジスタ中の値をインクリメントするだけでよく，GPU メモリへのアクセスが必要ないことから，従来の CSR に比べメモリアクセス回数の削減

が可能である．

14 行目から 18 行目は，不連続な非ゼロ要素を演算する部分であり，CSR の SpMV カーネルと同様の処理を行っている．不連続な非ゼロ要素に対し演算を行い，連続した非ゼロ要素に対する演算結果と合算している．最後に結果を各スレッドが担当していた行と同じベクトル中の行に格納し，終了する．

5.5 ELLPACK への RBP 法の適用 (RBP-ELL)

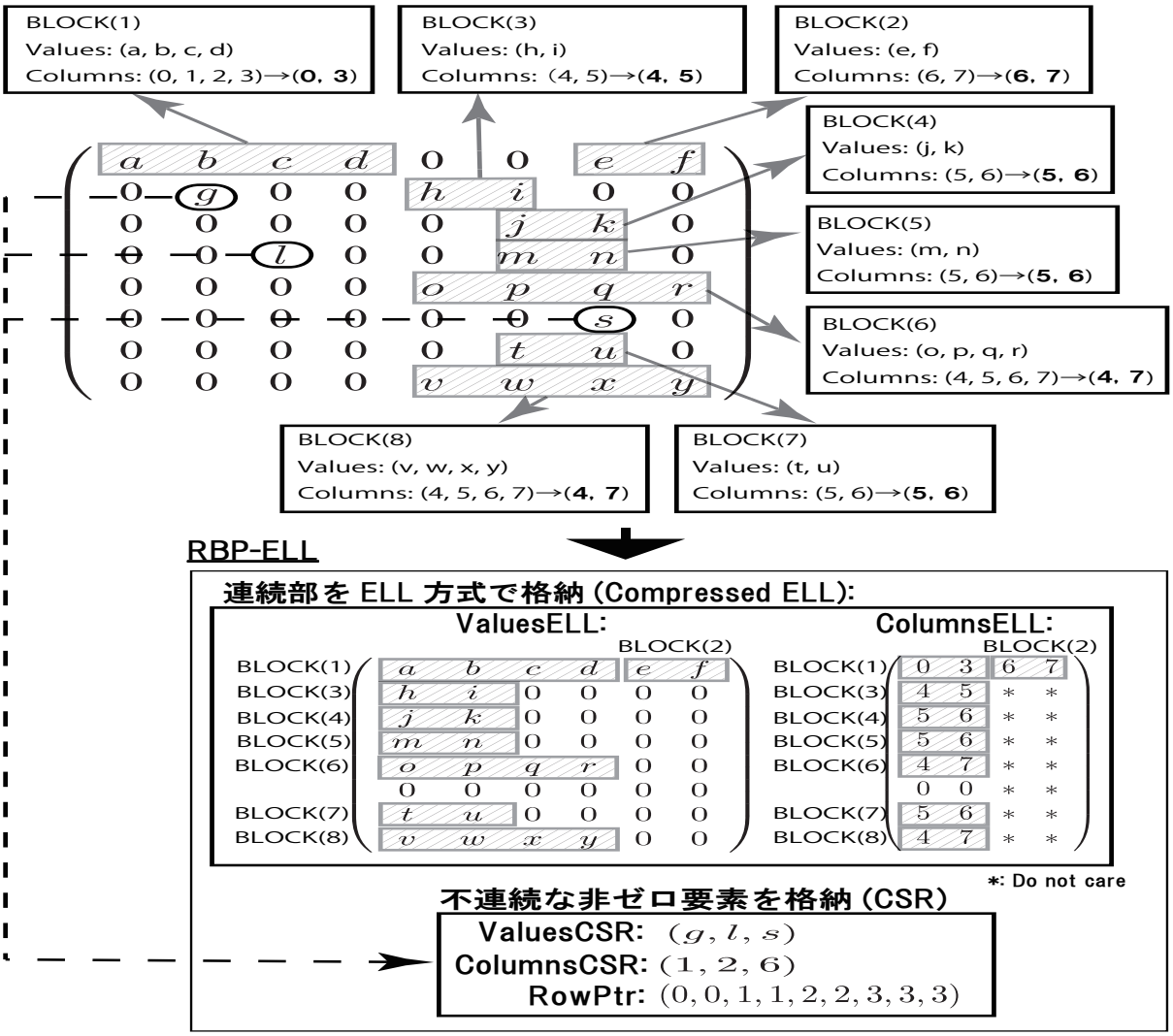

図 5.4: RBP-ELL

図 5.4 に RBP 法を施した ELL 格納方式，RBP-ELL の概念図を示す．ELL に RBP 法を適用する際も，Algorithm8 の 9 行目,「BLOCK を適用する疎行列格納方式で格納」以外は

RBP-CSR と同様である.「BLOCK を適用する疎行列格納方式で格納」において，ブロックの $Values$ と $Columns$ のデータを ELL の方式に則り，$ValuesELL$ と $ColumnsELL$ にインデックスの小さいブロックから格納していく．ここでは図 5.4 の Compressed ELL のように，各ブロックを 1 つの非ゼロ要素として扱い，格納する．RBP-CSR の場合と異なり，追加の Ptr 配列は必要としない．最後に CSR の処理 5 と同様に，不連続な非ゼロ要素を CSR 形式で格納し，終了する．ここで RBP-CSR の場合と同様に，不連続な非ゼロ要素の値を $ValuesCSR$ に，列番号は $ColumnsCSR$ に格納される．また，各行の最初の非ゼロ要素の情報が $ValuesCSR$ と $ColumnsCSR$ のどのインデックスに格納されているかを，$RowPtr$ に格納する．

式 (5.2) に RBP-ELL を用いて疎行列を格納した場合のメモリ使用量を示す．

$$\begin{aligned} MemUsage_{RBP-ELL} &= 8NK_{\mathrm{v}} + 4NK_{\mathrm{c}} \\ &+8N_{\mathrm{non}} + 4N_{\mathrm{non}} + 4(N+1)\ [byte] \end{aligned} \tag{5.2}$$

ここで，K_{v} は Compressed ELL の $ValuesELL$ 行列の列数，K_{c} は $ColumnsELL$ 行列の列数を示す．$8NK_{\mathrm{v}} + 4NK_{\mathrm{c}}$ は Compressed ELL 中の $ValuesELL$, $ColumnsELL$ が要とするメモリ使用量を示している．圧縮により K_{c} が小さくなるほど，メモリ使用量の削減が可能となる．$8N_{\mathrm{non}} + 4N_{\mathrm{non}} + 4(N+1)$ は図 5.3 中 RBP-ELL の $ValuesCSR$, $ColumnsCSR$, $RowPtr$ が必要とするメモリ使用量を示す．

Algorithm11 に RBP-ELL を用いた GPU 上での SpMV の疑似コードを示す．id はカーネル関数を実行する各スレッドの番号を示す．RBP-ELL では 1 スレッドが疎行列内の 1 行の計算を担当する．4,5 行目の $StartELL, EndELL$ には，連続した非ゼロ要素の先頭と末尾の要素の列番号がそれぞれ代入される．7 行目から 18 行目までの for 文では，Compressed ELL に対して処理を行う．for 文では，0 から $iMaxVal - 1$ まで j を増加させていく．$iMaxVal$ は RBP-ELL の ValuseELL 行列の列数を表している．ループ内では，8 行目のように，StartELL は必要なベクトルのインデックスを表現している．このベクトル $Vector[StartELL]$ と $Values$ 行列との掛け算を $TmpResult$ に足し合わせていくことで，SpMV の処理を行う．そして，9 行目にて RBP-CSR の時と同様に削除された列番号を復元するため，$StartELL$ をインクリメントする．その後，10 行目にて，$StartELL$ と $EndELL + 1$ が同値になったとき，新たな連続した非ゼロ要素の列番号を更新する (11,12 行目). また，14 から 16 行目では，スレッドが担当している行の全ての要素に対して処理が終了した，もしくはパディングである “0” を検知した場合に処理を打ち切る．そして，19 行目から 23 行目では，図 5.3 の不連続な非ゼロ要素を格納してある CSR を用いて SpMV 演算を行う．その後，連続部の演算結果が格納されている $TmpResult$ に不連続分の演算結果を足し合わせる．最後 24 行目で，各スレッドは自分の id と対応するベクトルのインデックスに解を格納し，SpMV の処理は終了となる．

Algorithm 11 SpMV code of RBP-ELL on GPU

```
1: Let id be thread ID (id: 0 to n-1)
2: Set Count, TmpReslut = 0
   /* iMaxVal is number of columns in ValuesELL */
3: Set iMaxVal
   /* Start point of consecutive non-zero elements */
4: Set StartELL to ColumnsELL[id][Count]
   /* End point of consecutive non-zero elements */
5: Set EndELL to ColumnsELL[id][Conut + 1]
6: Count = Count + 2
   /* Calculation of ELL part */
7: for j = 0 to iMaxVal − 1 do
8:    TmpResult+ = ValuesELL[id][j] ∗ Vector[StartELL]
9:    StartELL + +
10:   if StartELL == EndELL + 1 then
11:      Set StartELL to ColumnsELL[id][Count]
12:      Set EndELL to ColumnsELL[id][Conut + 1]
13:      Count = Count + 2
         /* iMaxCol is number of columns in ColumnsELL */
14:      if Count > iMaxCol or StartELL == EndELL then
15:         break
16:      end if
17:   end if
18: end for
   /* Calculation of CSR part */
19: Set StartCSR to RowPtr[id]
20: Set EndCSR to RowPtr[id + 1] − 1
21: for j = StartCSR to EndCSR do
22:    TmpResult+ = ValuesCSR[j] ∗ Vector[ColumnsCSR[j]]
23: end for
24: Result[id] = TmpResult
```

5.6 評価実験概要・環境

本節より RBP 法の評価実験を行う．RBP 法は疎行列からの変換時間を重視した手法であり，GMRES において変換時間が支配的にならないことを確認することが重要である．広い範囲に対して圧縮を行う PatComp に比べて，狭い範囲での圧縮を行う RBP 法がメ

モリ使用量の削減率，SpMV 演算時間がどのように影響するか確認する．

表 5.1: Expriment condition

	Specification
OS	Ubuntu 16.04
CPU	Intel Core i7 6700K @ 4.0 GHz
GPU	NVIDIA Tesla V100 @ 1.38 GHz
device memory	16 GB
device memory bandwidth	900 GB/s
CUDA core	5,120
CUDA	CUDA 10.1
Compiler	gcc-4.4.7

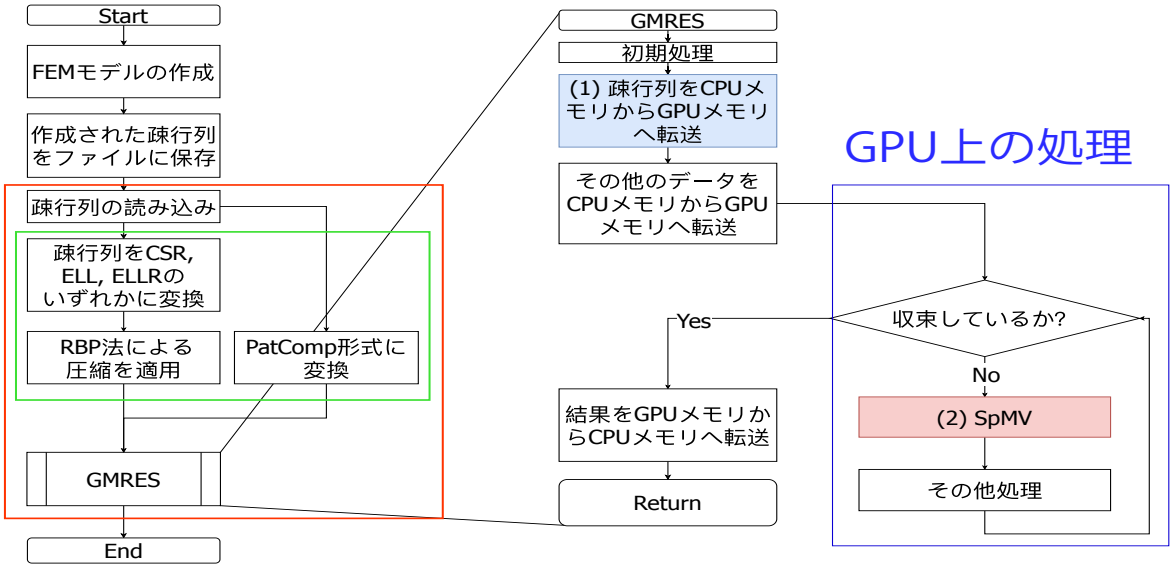

図 5.5: Model of Calculating GMRES with RBP and PatComp

RBP 法の評価実験で使用する GPU サーバ，疎行列は 4.6 で使用した条件と同じである．再度表 5.1，表 5.2 に実験環境と使用した疎行列を示す．

また，表 5.2 中の N,N_z,N_{non},K，K_{v},K_{c} は，式 (3.1) から (5.2) で使用されている変数に対応している．本実験にて使用する疎行列格納方式は，CSR, RBP-CSR, ELL, RBP-ELL, ELL-R，RBP-ELL-R，PatComp である．RBP-ELL-R は，RBP-ELL に各行の非ゼロ要素数を示す配列を追加しただけである．ELL の演算性能を高めた ELL-R に RBP 法を適用することで，RBP 法の演算性能への影響も確認する．

表 5.2: Sparse Matrices for the expiments

Name of matrix	N	N_{z}	N_{non}	K	K_{v}	K_{c}
cant	62,451	4,007,383	87,657	78	78	28
rma10	46,835	2,374,001	447	145	145	40
consph	83,334	6,010,480	52,325	81	81	42
parabolic_fem	525,825	3,674,625	2,081,987	7	7	6
pwtk	217,918	11,524,432	4,049	180	180	28
thermal2	1,228,045	8,580,313	5,192,466	11	10	8
af_shell9	504,855	17,588,845	0	40	40	10
F1	343,791	26,837,113	9,300	435	435	162
nd24k	72,000	28,715,634	887,233	520	509	126
dielFilterV2real	1,157,456	48,538,952	18,745,748	110	87	56
Cube_Coup_dt0	2,164,760	124,406,070	0	68	68	34
Bump_2911	2,911,419	127,729,899	86,208	195	195	68
Queen_4147	4,147,110	316,548,962	1	81	81	54
UT-Heart1	82,047	3,423,519	0	63	63	40
UT-Heart2	130,595	6,954,413	969,595	124	111	50

図 5.5 は実験で使用する，FEM にて生成された連立一次方程式（疎行列）に RBP 法. PatComp 法を適用し，GMRES で解くことを想定したモデルを示す．赤枠より上の処理,「FEM モデルの生成」,「作成された疎行列をファイルに保存」で保存された疎行列に, Florida Sparse Matrix Collection またはアプリケーション例として加えた 2 つの疎行列を使用することで評価を行う．評価対象は，図 5.5 中の赤枠，つまり GMRES 処理の開始から終了までである．GPU による評価を行うために，NVIDIA が提供する CUDA10.1 を使用し，各疎行列格納方式を用いた SpMV を組み込んだ GMRES プログラムを記述した. 疎行列ファイルの読み込みから RBP 法，PatComp の適用，GMRES の初期処理までが CPU 上で処理し，その後は GPU 上での処理となる．図 5.5 の青枠で囲まれた部分が GPU 上で実行される．データ転送時間，GPU 演算時間は，CUDA のイベント変数を用いて, “cudaEventRecord()”,“cudaEventElapsedTime()” により測定を行った．また GPU による高速化の有効性を示すため，CPU 版の GMRES との比較を行う．CPU 版の GMRES は図 5.5 の処理全てを CPU 上で行う.

5.7 節では，RBP 法，PatComp 法をそれぞれ用いて疎行列を格納した際に必要となるメモリ使用量を評価し，5.8 節では図 5.5 内の SpMV(赤ブロック) の SpMV を 1 回実行した際の演算時間を評価する．最後に 5.9 節にて，図 5.5 内の赤枠全体の演算時間の評価を行う.

5.7 各疎行列格納方式のメモリ使用量の評価

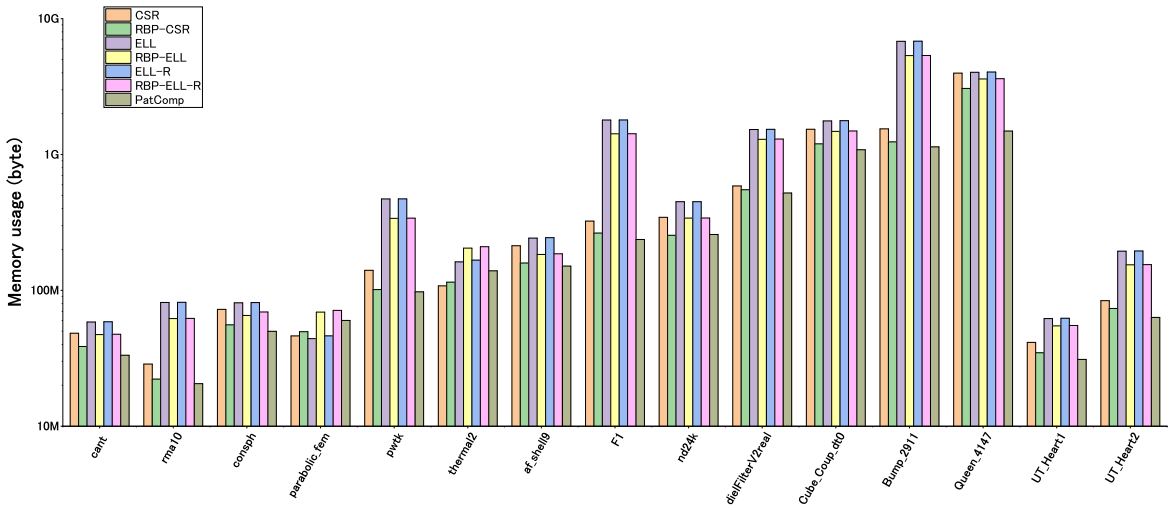

図 5.6: Memory usage of RBP formats

本節では RBP 法を使用することにより，従来の疎行列格納方式と比べメモリ使用量の削減が達成されたか確認する．図 5.6 に，各疎行列格納方式を用いて，表 5.2 の各疎行列を格納した際のメモリ使用量を示す．この RBP-ELL-R 以外の疎行列格納方式のメモリ使用量は，式 (3.1) から (5.2) を用いて導かれた値である．また，RBP-ELL-R のメモリ使用量は以下の式 (5.3) から導かれた値となっている．ELL-R は ELL に要素数が行数に等しい配列が追加されているため，RBP-ELL のメモリ使用量を求める式 5.2 に $4N$ を追加した形となっている．

$$\begin{aligned} MemUsage_{RBP-ELL-R} &= 8NK_{\mathrm{v}} + 4(NK_{\mathrm{c}} + N) \\ &\quad +8N_{\mathrm{non}} + 4N_{\mathrm{non}} + 4(N+1)\ [byte] \end{aligned} \tag{5.3}$$

CSR，ELL，ELL-R と RBP-CSR，RBP-ELL，RBP-ELL-R のメモリ使用量を比較するとそれぞれ，15 個中 parabolic_fem, thermal2 を除く 13 個の行列において，RBP 法を適用した格納方式の方が少ない結果となった．

RBP-CSR と CSR を比較すると，pwtk において，最大の 27.9%のメモリ使用量削減に成功した．また 15 個の疎行列における，メモリ使用量の平均削減率は 16.6%であった．この結果から，FEM で生成される行列には連続した非ゼロ要素が多く存在することが確認できる．

RBP-ELL と ELL を比較すると，pwtk において，最大の 28.0%のメモリ使用量削減に成功している．また RBP-ELL を用いて疎行列を格納することで，15 個の疎行列のメモリ使用量を平均 11.5%削減した．

RBP-ELL-R が最もメモリ使用量を削減できた疎行列は RBP-ELL と同じく pwtk であり，削減率は 28.0%であった．また平均メモリ使用量削減率は，11.4%であった．

結果として RBP 法の適用により，既存の疎行列格納方式のメモリ使用量を 7 個以上の疎行列において，20%以上の削減に成功した．大きな削減率となった疎行列の特徴として，連続した非ゼロ要素が各行に多く存在することがあげられる．また医療分野のアプリケーション例として評価に使用した UT-Heart1，UT-Heart2 においても，RBP-CSR でそれぞれ 16.1%，12.4%，RBP-ELL と RBP-ELL-R でそれぞれ 11.6%，20.6% のメモリ使用量の削減を達成した．巨大な疎行列である Cube_Coup_dt0，Bump_2911，Queen_4147 においてもメモリ使用量の削減に成功しており，BRP 法の有効性を示した．また更に使用した疎行列の問題を細分化，大規模化させた場合でも非ゼロ要素のパターンは大きく変化しないと考えられ，RBP 法が有効であると考察している．Bump_2911 では，RBP 法を ELL，ELL-R に適用することでメモリ使用量を 6GB 以下に削減することに成功した．これにより GeForce 1060 の 6GB モデルでは困難であったシミュレーションが可能となり，Tesla 等の高い GPU の代替として低コストな GPU を使用することも可能となる．

しかしながら PatComp と同様に，parabolic_fem と thermal2 の 2 つの行列では，RBP 法を用いても既存の疎行列格納方式のメモリ使用量を削減できず，逆にメモリ使用量が大幅に増加した．これは，parabolic_fem や thermal2 の行列内には，連続した非ゼロ要素が少ないことと，連続した場合でも連続数が少ないことが原因である．

parabolic_fem では，非ゼロ要素の最大連続数が 2 であり，RBP 法による効果がない．それどころか，不連続な非ゼロ要素を格納するのに CSR を使用するため，この分のメモリ使用量が増大した．

thermal2 の非ゼロ要素の連続数 7 の行が存在し，その行では RBP 法により 7 個の非ゼロ要素の列番号を二つの値で表現することで，大幅に列番号を削減できる．しかしながら 2 連続の非ゼロ要素が複数存在する行も thermal2 には存在する．そのような行に対しては RBP 法の効果はなく，全ての非ゼロ要素の列番号を格納することになる．$ColumnsELL$ では最も非ゼロ要素が多い行の要素数に列数が依存するため，大幅に列番号を削減できる行があったとしても，行ごとの削減率にばらつきがある場合には圧縮効果が小さくなる．また thermal2 は疎行列内の非ゼロ要素数 8,580,313 のうち 5,192,466 が不連続な非ゼロ要素であり，連続した非ゼロ要素が少ない．連続した非ゼロ要素が少なく，削減できた非ゼロ要素の列番号が少ないため，RBP-CSR では Col_ptr を追加した分のメモリ使用量による増加の方が大きくなった．

今回評価対象として Florida Sparse Matrix Collection を用いたのは，従来の CSR や ELL との比較のためである．現実にはこのような連続した非ゼロ要素が少ない問題は稀であるため，提案手法は多くの場合有効であると考えている．仮に連続した非ゼロ要素が少ない問題があったとしても，FEM モデルの節点番号を振りなおすことにより，連続した非ゼロ要素数を増加させる方法も考えられる．

また 4 章で提案した PatComp と RBP-CSR を比較すると，平均で 4.3%PatComp の方がメモリ使用量が低くなった．PatComp は RBP 方に比べ広い範囲を短いインデックス

で置き換えることから，メモリ使用量削減率が高くなったと考えられる．しかしながらRBP 法は変換に必要な計算量が少ないながらも，CSR に比べ最大 27.9%の削減に成功していることから，時間ステップ毎に変換が必要な問題にとって，有用性は高いと考える．

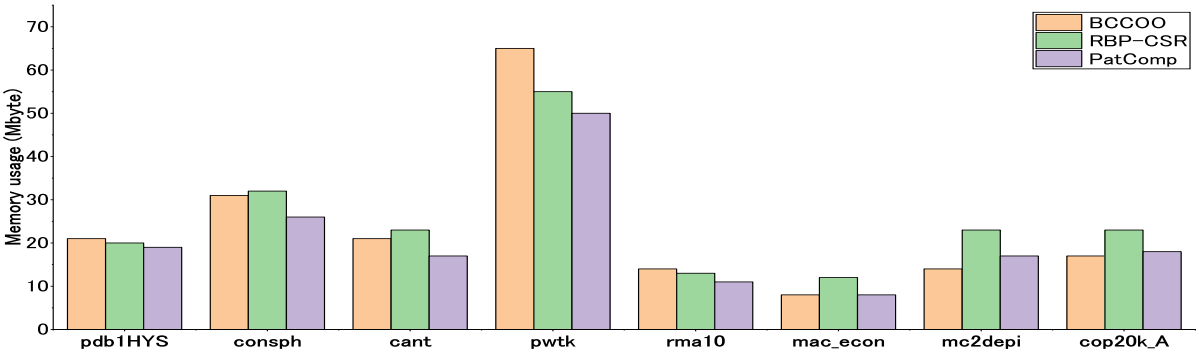

図 5.7: Memory usage comparision with BCCOO

また図 5.7 に RBP-CSR と BCCOO のメモリ使用量の比較を示す．BCCOO は RBP 法と同様，非ゼロ要素の位置情報を削減する手法である．今回 BCCOO のメモリ使用量は論文中に記載されているメモリ使用量の値を使用している．また比較に使用する疎行列は FEM に限定せず，BCCOO に記載されている疎行列のうち 7 個の疎行列を選択した．RBP-CSR のメモリ使用量を BCCOO と比較すると pdb1HYS，pwtk 以外で増加した結果となった．FEM の疎行列ではない mac_econ，mc2depi では大幅なメモリ使用量の増加が見られた．BCCOO では，複数行，複数列で構成される固定サイズのブロックを用いて疎行列を分割し，各ブロックに基準となる位置情報を一つ割り当てる．そのため適切にブロックサイズを選択することが出来れば，1 行中の連続した非ゼロ要素を対象としている RBP 法より大幅なメモリ使用量の削減が可能である．しかし固定サイズのブロックを用いることから，ブロックの範囲内に非ゼロ要素が存在しない部分に関してはゼロ要素をパディングして補う必要がある．この場合にはメモリ使用量の削減率が低くなったり，最悪の場合メモリ使用量が増加することがある．そのため BCCOO では複数のブロックサイズで疎行列を格納し，その中でメモリ使用量が最も少ないサイズを選択する．この最適化により多くの疎行列で RBP 法よりメモリ使用量が多少少ない結果となった．

時間発展ごとに疎行列の形式が変わる数値シミュレーションでは，時間ステップごとに変換を行う必要があるため，疎行列格納方式への変換時間が重要となる．しかしながら BCCOO は多くの最適化を行う必要があることから変換時間が多く必要となる．一方 RBP 法は各非ゼロ要素の列番号を一度参照するだけで変換することができることから変換が高速である．RBP 法の変換時間の評価は 5.9 節にて行う．

RBP 法は BCCOO に比べ変換時間が短いながらも，mac_econ，mc2depi を除く FEM で生成された疎行列において平均メモリ使用量の差は 3.4%に抑えられている．

5.8 GPU 上での SpMV 演算時間の評価

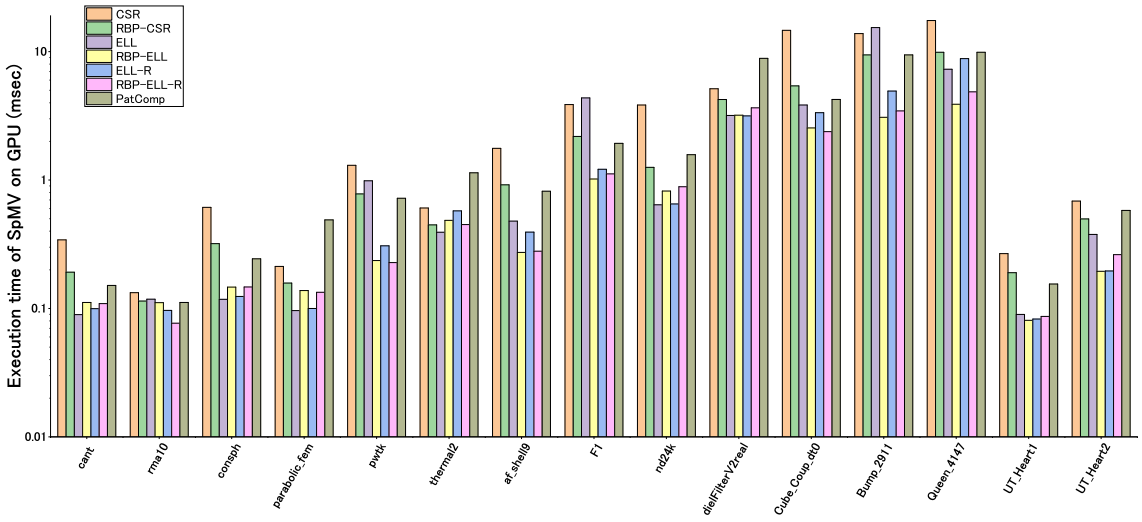

図 5.8: Execution time of SpMV using RBP format on GPU

本節では，RBP 法を各疎行列格納方式に適応することによる SpMV 演算時間への影響を評価する．

図 5.8 に各疎行列格納方式を用いた SpMV の演算時間を示す．GMRES をはじめとする反復法では，収束まで SpMV が複数回実行されるが，ここでは比較のため SpMV カーネルの 1 回あたりの演算時間を示す．CSR，ELL の SpMV カーネルはそれぞれ Algorithm4, 5 を元に CUDA を用いて記述した．

RBP-CSR では全ての疎行列で，CSR に比べ SpMV 演算時間が短縮された．RBP-CSR が最も演算時間の削減に成功した nd24k では，CSR の SpMV 演算時間に比べ 3.06 倍の高速化を達成した．また 15 個の疎行列の平均では，1.82 倍の高速化となった．CSR では連続している非ゼロ要素の数だけ，非常に低速なグローバルメモリに対してアクセスする必要があり，大きなボトルネックとなる．これに対し，RBP-CSR を用いた SpMV カーネル（Algorithm10）では，連続した列番号に対し，最初と最後の列番号の 2 つのみをグローバルメモリから読み出し，レジスタに格納している．そのため，グローバルメモリアクセス回数が削減され，演算時間も削減された．

RBP-ELL では 15 個中 9 個，RBP-ELL-R では 15 個中 8 個で既存疎行列格納方式より SpMV 演算時間が短くなった．RBP-ELL は，F1 において最大の 4.28 倍の高速化に成功している．全体の平均高速化率は 1.57 倍である．RBP-ELL-R において，最も高速化に成功した行列は Queen_4147 であり，1.81 倍の高速化に成功している．平均高速化率は 1.14 倍である．高速化を達成した行列では，RBP-CSR と同様にグローバルメモリへのアクセス回数削減によるアクセス時間の短縮が演算性能向上に貢献したと考えられる．

しかしながらRBP-ELLで6個，RBP-ELL-Rで7個の疎行列にて演算時間の増加が見られた．ELLやELL-Rは3.4.4で述べたようにパディングを用いることでGPU上での最適なメモリアクセスを実現している．しかしRBP法による圧縮によりELL, ELL-Rで実現されていた各スレッドの連続したアドレスへのアクセスが崩れてしまったことが1つの原因と考えられる．各Threadが処理する連続した非ゼロ要素の連続数が異なる場合，$ValuesELL$のアクセスされる要素数もThreadごとに異なる．そのため各Threadが次の非ゼロ要素を処理しようとした際，非ゼロ要素の値をそれぞれ$ValuesELL$の離れた列のインデックスへアクセスする必要が出てくる．離れたインデックスへのアクセスが発生すると図3.10のような連続したアドレスへのアクセスは困難となり，演算時間が増加する．またメモリ使用量が増加したparabolic_fem，thermal2ではメモリアクセス回数が削減されないだけでなく，追加したCSRへのアクセス回数が増加し，演算時間が大幅に増加した．よってRBP-ELL，RBP-ELL-Rを用いて演算性能が向上する疎行列の条件として，各行に存在する複数の連続した非ゼロ要素の連続数が，それぞれほぼ等しいことが挙げられる．同じ連続数で連続する非ゼロ要素を処理する場合にはELLと同様の効率的なメモリアクセスをRBP-ELLでも実現することが可能である．

またRBP-CSRはPatCompとほぼ同等のSpMV演算時間となった．しかしながらPatCompの方がメモリ使用量の削減に失敗した疎行列においてSpMV演算時間が大幅に増加することが判明した．メモリ使用量が増加した分，不要なインデックスへのアクセスが増加したことから演算時間も増加したと考えられる．

さらに詳細なCache hit rateやMemory throughput等の詳細な解析を次節よりPatCompも含めて比較する．

5.8.0.1 GPU Speed of light (SOL)

本節では，GPUの計算資源の使用度を示すGPU Speed of light (SOL)について評価する．図5.9にGPU上の演算器Streaming multiprocessorの使用効率“SOL of SM”を示す．また図5.10にGPU上のメモリ使用効率を示す“SOL of Memory’を示す．

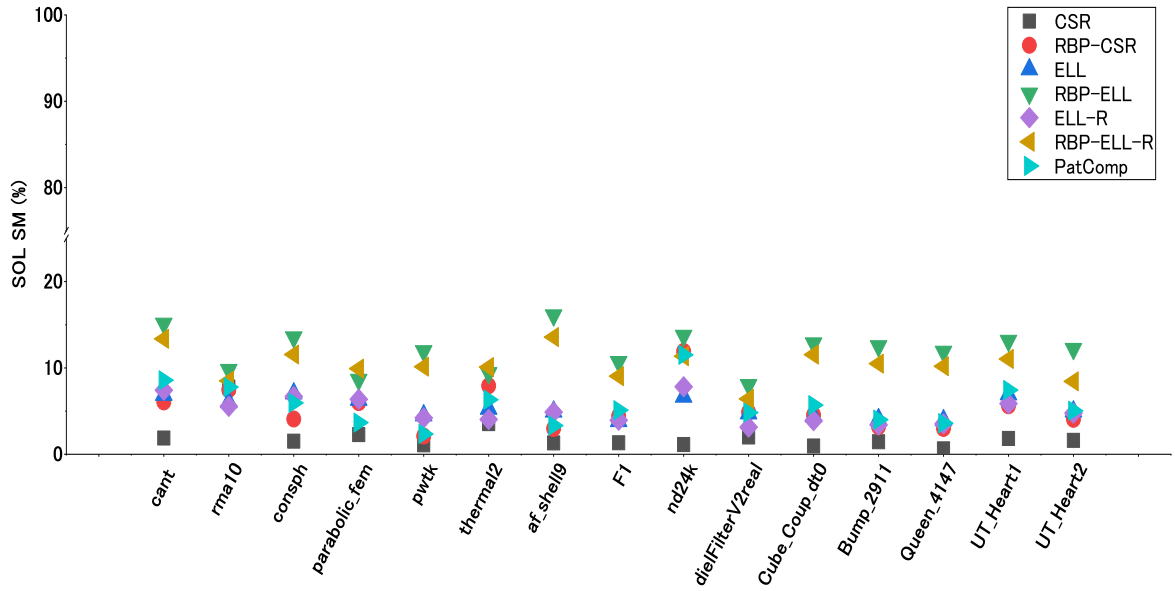

図 5.9: SOL of SM

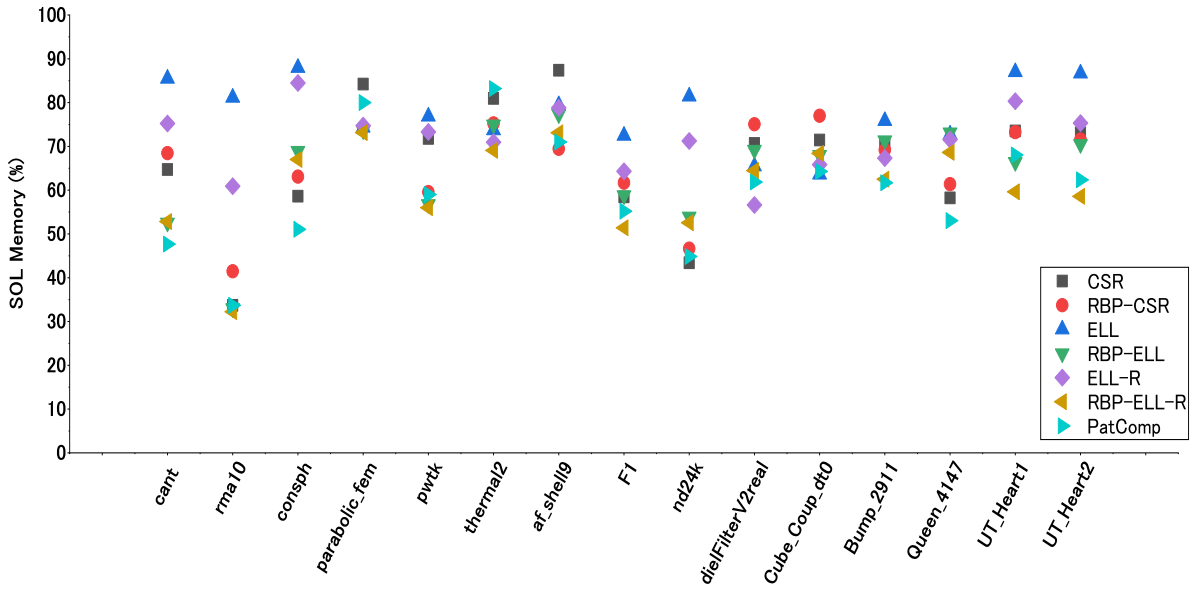

図 5.10: SOL of Memory

まず全ての疎行列格納方式において，“SOL of SM” に比べ “SOL of Memory” の割合が大きくなっていることから，SpMV はメモリバウンドな処理であることがわかる．そのため，メモリアクセス回数を削減し，SM での演算回数を増やすことが SpMV 演算性能向上につながる．

RBP 法，PatComp 法では列番号を削減することでメモリアクセス回数を削減し，その分レジスタの値をインクリメントすることで列番号を復元した．そのため，RBP-CSR, RBP-ELL，RBP-ELL-R の “SOL of SM” は既存の CSR，ELL，ELL-R に比べ高くなっている．また RBP 法を用いても演算時間が短くならなかった，もしくは小さい高速化に留まった疎行列，parabolic_fem，thermal2，dielFilterV2real の “SOL SM” は他の疎行列に比べ小さな値となっている．これは疎行列内の連続した非ゼロ要素が少なく，1 つあたりの連続数が短いことから，インクリメントで列番号を復元する処理が少なくなったことによる結果である．

“SOL of Memory” では，ELL，ELL-R が高くなっている．これは ELL,ELL-R は GPU に適したメモリアクセスが行えるためである．それに比べ RBP 法を用いた RBP-ELL, RBP-ELL-R は低くなっている．圧縮により ELL や ELL-R のメモリアクセス効率が低下したためだと考えられる．

5.8.0.2 Warp cycles per issued instruction

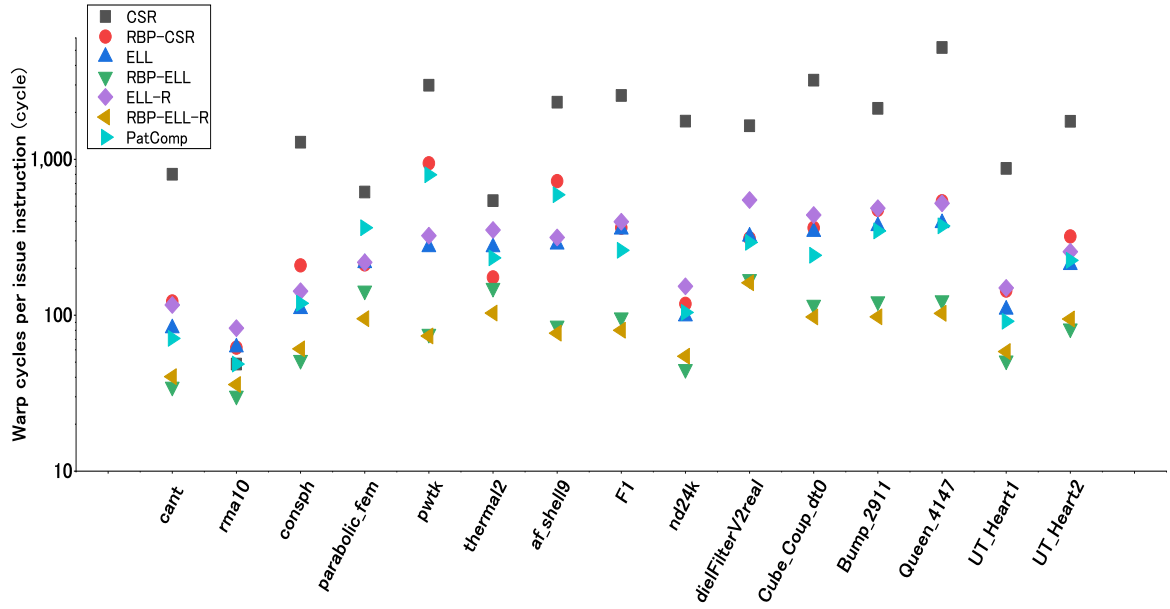

図 5.11: Warp cycles per issued instruction

図 5.11 に 1 命令あたりの Warp cycles を示す．Warp cycles が大きい場合，メモリアクセス等の様々な処理によるストールが頻発して演算にかかるクロック数が増加したことを意味する．

まず多くの疎行列において SpMV 演算性能が最も悪い CSR の Warp cycles は，他の疎行列格納方式より非常に大きくなっている．CSR は不連続なアドレスへのメモリアクセスが多く，メモリアクセス回数が増加することから，メモリアクセス待ちのストールが多

く発生し，1 命令あたり Warp cycles が非常に大きくなった．CSR に比べ RBP-CSR の 1 命令あたりの Warp cycles が大きく減少している．これは列番号数の削減を行うことにより CSR におけるボトルネックとなるメモリアクセス回数が現象し，メモリアクセス待ちのストールも削減された．

RBP-ELL，RBP-ELL-R と ELL，ELL-R を比較し，全ての疎行列において RBP 法の方が少ない Warp cycles で 1 命令が実行できることが図 5.11 からわかる．この削減の理由は CSR と RBP-CSR の場合と同じくメモリアクセス待ちの削減による効果だと考えらえる．

またメモリ使用量削減を第一に考えた疎行列格納方式，CSR，RBP-CSR，PatComp の中で，PatComp が多くの疎行列で最も良い結果となっている．これは複数の Thread が同じデータを読み出すことによる Cache hit rate の向上によるメモリアクセスに必要となるストール数減少が要因と考えている．

5.8.0.3 Cache hit rate

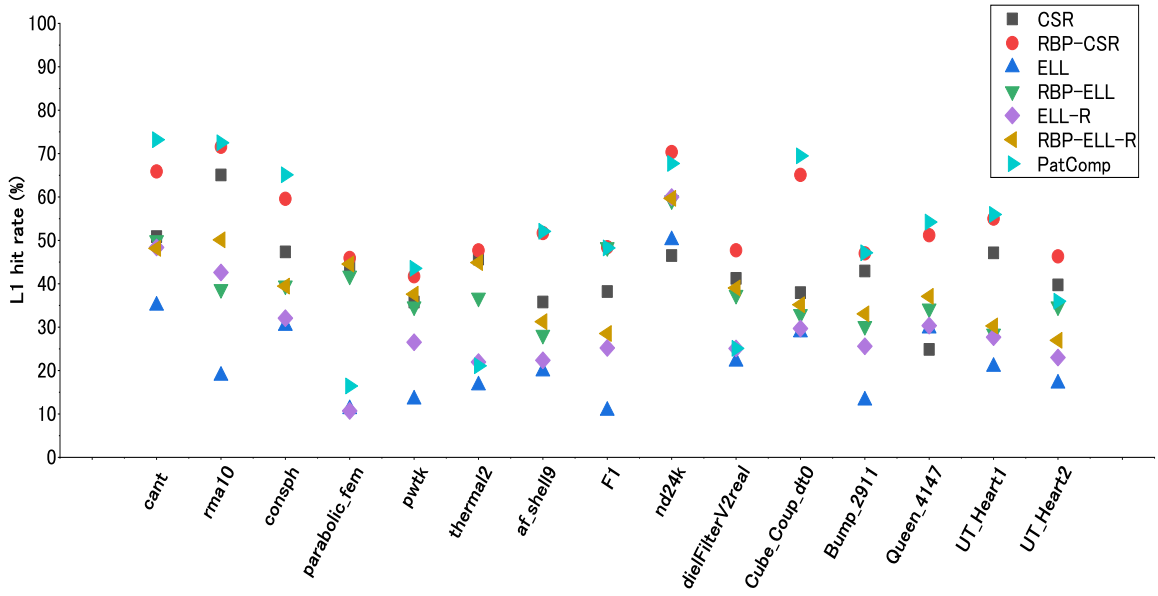

図 5.12: Hit rate of L1 cache on GPU

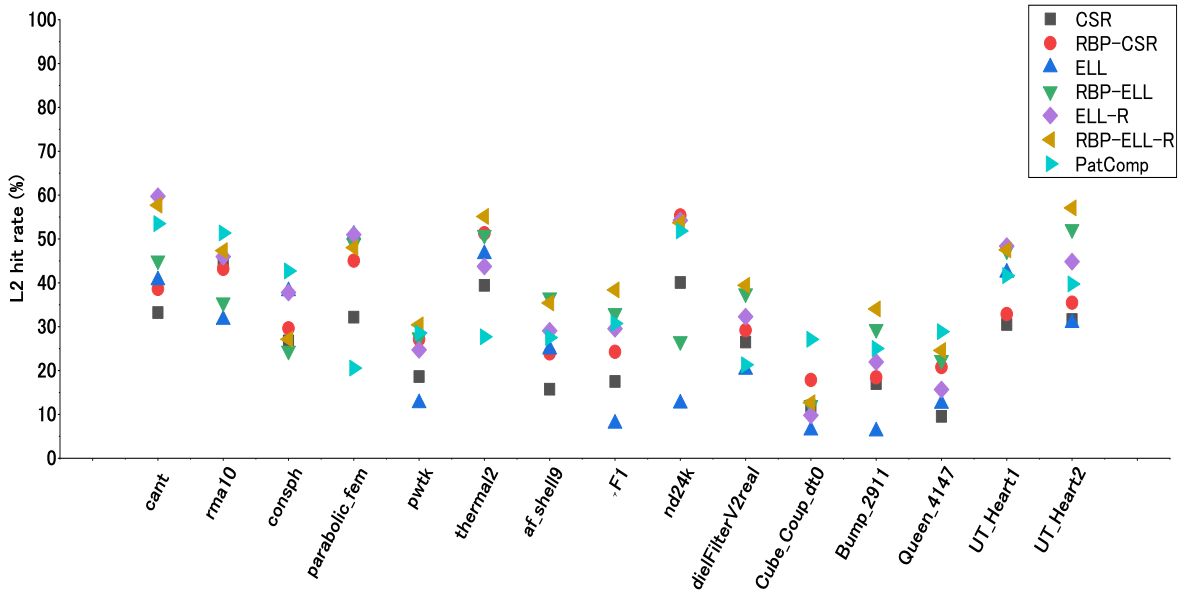

図 5.13: Hit rate of L2 cache on GPU

本節において各疎行列格納方式を用いて SpMV を実行した際の L1 Cache hit rate，L2 Cache hit rate について評価を行う．また PatComp は複数の行の非ゼロ要素の位置を判別するために複数の Thread が同じパターンを使用することから，Cache hit rate の向上が期待できる．そのため本節では PatComp の Cache hit rate にも注目し，評価を行う．

図 5.12 に GPU 上での L1 Chace hit rate を示す．parabolic_fem，thermal2，nd24k，dielFilterV2real，UT_Heart2 を除く 11 個の疎行列において PatComp の Cache hit rate が最も高くなっている．少ないパターンで多くの行を表すことができる疎行列ほど，L1 Cache hit rate が高くなっている．L1 Cache は図 3.3 に示すように，各 SM のいくつかの CUDA core ごとに搭載されている．この結果から疎行列のパターンを考慮することで，SpMV の演算性能向上に繋がると考えられる．多くの疎行列において PatComp が目標とした，L1，L2 Cache hit rate の向上を達成した．PatComp がメモリ使用量を大幅に削減した疎行列において Cache hit rate は向上する傾向にある．

PatComp を用いても L1 Chache hit rate が向上しなかった parabolic_fem，thermal2，nd24k，dielFilterV2real，UT_Heart2 は，連続した非ゼロ要素が少なく不連続な非ゼロ要素が多い，または全ての行を表現するために多くのパターンが必要となる行列である．パターン数が多くなるとキャッシュを有効に活用できず，SpMV の演算性能も低下することが分かる．

また図 5.13 に L2 Cache hit rate を示す．L2 Chace hit rate に関しても PatComp は，CSR，RBP-CSR に比べ高くなる傾向にある．L1 Cahce で hit rate が向上しなかった疎行列では，L2 Cache においても同様に hit rate が向上していない．

また RBP 法を用いることで既存の CSR，ELL，ELL-R に比べ，Cache hit rate が向上

した．この Cache hit rate の向上が，RBP 法の演算性能向上の要因の一つになったと考えらえる．

5.8.0.4 Memory throughput

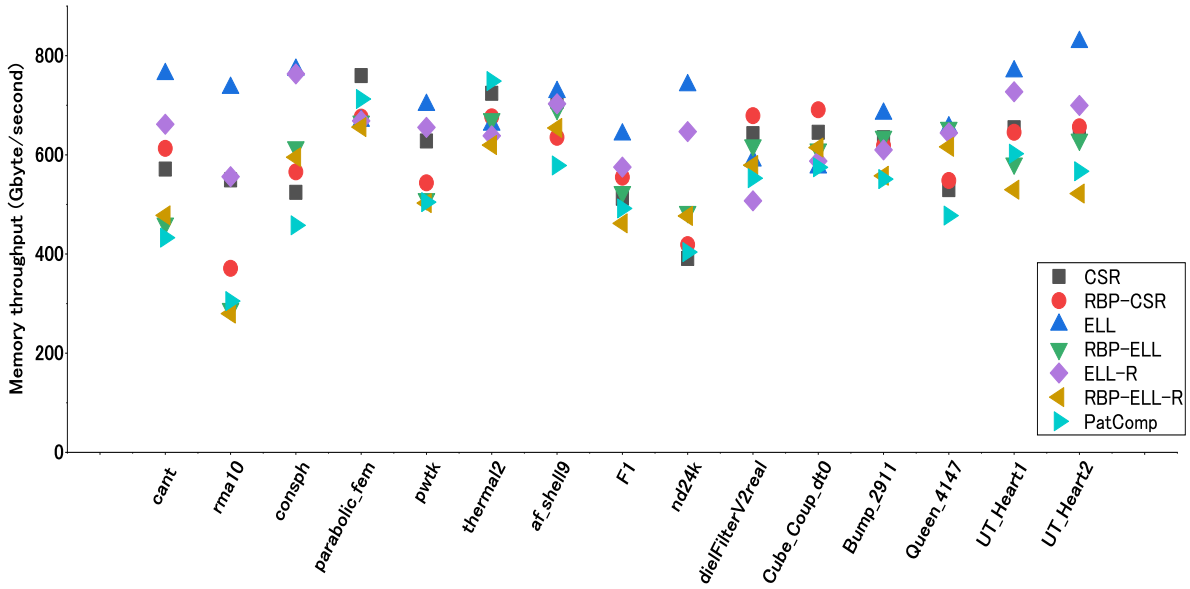

図 5.14: Memory throughput of each storage formats

図 5.14 に各疎行列格納方式を用いて SpMV を実行した際の Memory throughput を示す．Memory throughtput は，全体の Thread がデバイスメモリから 1 秒あたりに読み出したデータ量 (Gbyte) である．

15 個中 10 個の疎行列において，ELL を用いた場合の Memory throughtput が最も高く，ELL に続いて ELL-R の Memory throughput が高い．これは ELL，ELL-R を用いることにより同一 Warp 内の Thread が連続したメモリアドレスにアクセス可能であることが要因である．連続したメモリアドレスにアクセスすることで，全ての Thread が必要とするデータを一度に読み出し可能である．そのため ELL や ELL-R は時間当たりの読み出し可能なデータ量は CSR に比べ，非常に多くなる．

また RBP-ELL や RBP-ELL-R では ELL，ELL-R に比べ，いくつかの疎行列において Memory throughput が低くなっている．5.8 節の評価実験において，このメモリアクセス効率の低下がいくつかの疎行列の SpMV 演算性能低下の原因であると考えらえる．

PatComp は 12 個の疎行列において RBP-CSR より Memory throughput が低くなった．しかしながら PatComp は Cache hit rate が高いため，デバイスメモリからのスループットが低くなっても，RBP-CSR より多くの疎行列で演算性能が高くなったと考えらえる．

RBP-CSR では多くの疎行列にてメモリ使用量を削減し，全ての疎行列にて演算性能の向上を達成した．RBP-ELL，RBP-ELL-R では演算性能の低下が見られる疎行列も存在したが，多くの疎行列においてメモリ使用量の削減を達成し，演算性能も既存手法と遜色ない結果となった．この結果からこれまでに使用されていた CSR や ELL，ELL-R に RBP 法を適用することでメモリ使用量を削減し，さらに大規模かつ高精度な数値シミュレーションが可能となった．

5.9 GMRES による演算時間の評価

本節にて各疎行列格納方式で格納された疎行列を，反復法の 1 種である GMRES 内の SpMV に適用した際，全体の演算時間へどのように影響するか検証する．RBP 法への変換時間を含めた GMRES の演算時間が，既存の手法を用いた場合に比べて同等以上であることを確認する．また最後に PatComp に比べ，疎行列から RBP 法への変換時間がどの程度短縮されているか比較する．

5.9.1 各疎行列格納方式を用いた GMRES の演算時間

本評価実験では GPU を用いて GMRES を高速化した場合と，全ての処理を CPU で行った場合の演算時間を示している．これは GPU が反復法，そして SpMV の高速化に有効であることを確かめるためである．

5.9.1.1 GPU を用いた GMRES の演算時間

まず GPU を用いた GMRES の演算時間の評価を行う．図 5.15 に，図 5.5 の赤枠で表されている，連立一次方程式を GMRES を用いて解くために必要となる処理時間を示す．縦軸は対数軸となっている．内訳として，図 5.5 における (1) 疎行列の CPU メモリから GPU メモリへの転送時間，(2)GPU 上での SpMV 演算時間の合計，およびその他の処理時間を，それぞれ緑，紫，黄色で表した．その他の処理時間 (Other) は、GMRES(図 5.5 の赤枠部分) から各処理時間を除いた残りの時間である．また図 5.5 の緑枠で示す各疎行列格納方式への変換時間をオレンジ色で示した．データ転送時間は各疎行列格納方式で格納された疎行列の転送時間のみを示している．行列本体以外のデータは疎行列格納方式に関わらず一定であるため，その他の処理時間に含めている．今回，GMRES の最大反復回数は，最大 1000 回としている．数値シミュレーションの必要とする精度等により，必要な反復回数が異なるため，今回は反復回数 1000 回での GMRES 演算時間を評価する．例外として UT_Heart2 は 277 回の反復で収束したため，その時点での GMRES 演算時間を示す．

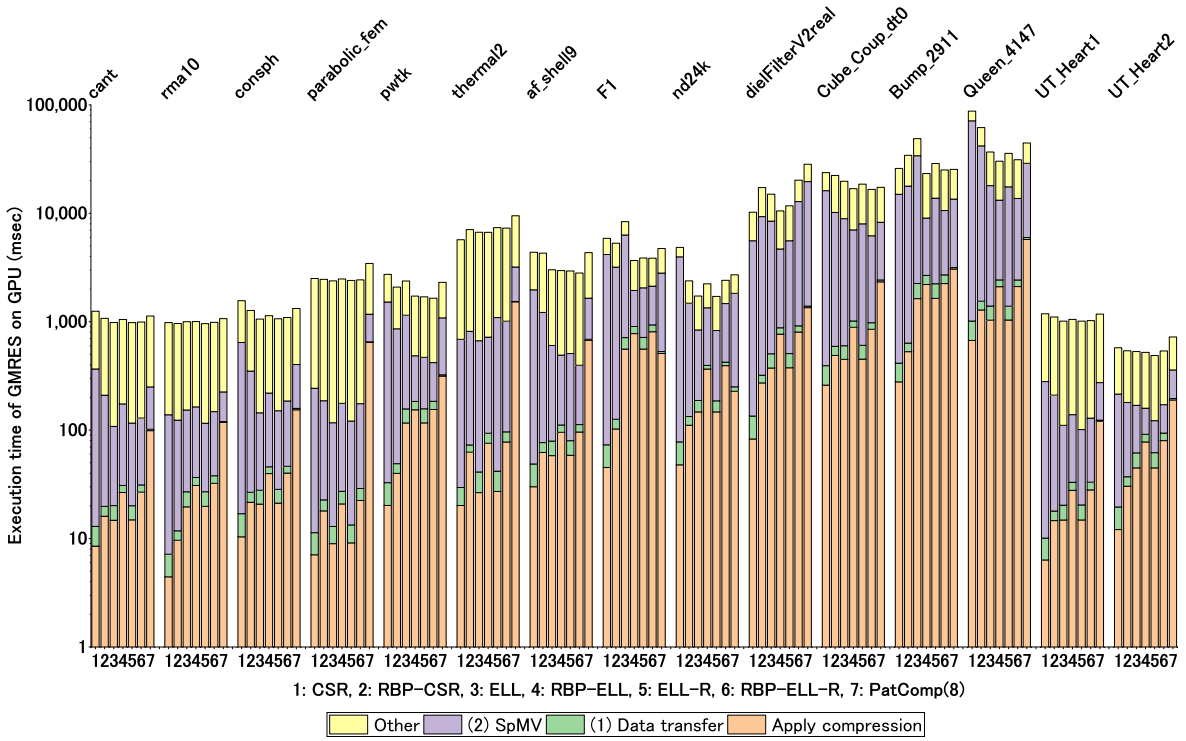

図 5.15: Execution time of GMRES on GPU

連立一次方程式の求解中で，各疎行列格納方式への変換，疎行列のデータ転送は 1 回のみであり，SpMV の実行は反復毎に行われるため，全体の処理時間に対する SpMV 演算時間の割合が，図 5.15 から多くの疎行列において高くなっていることが確認できる．

疎行列格納方式への変換を含まない GMRES 全体の演算時間を，既存格納方式 CSR, ELL，ELL-R と RBP 法を適用した格納方式 RBP-CSR，RBP-ELL，RBP-ELL-R で比較すると，RBP 法を適用した場合の方が平均で 4.1%, 13.2%, 3.1%短くなった．

疎行列格納方式への変換時間を含めた場合でも，CSR を用いた GMRES と RBP-CSR を用いた GMRES の演算時間を比較すると，RBP-CSR を用いた演算時間の方が平均で 3.1%短くなった．同様に RBP-ELL を用いた GMRES は，ELL を用いた場合に比べ，平均 10.4%演算時間が短くなった．RBP-ELL-R を用いた GMRES は，ELL-R を用いた場合に比べ，平均 1.5%演算時間が短くなった．RBP 法の目的である高速な変換が達成されたことにより，変換時間が GMRES のボトルネックとなっていないことを確認した．時間発展に伴い疎行列の形が変化する問題では，時間ステップごとに GMRES をはじめとする反復法を解く必要がある．その度に疎行列の変換処理が必要となることから，RBP 法のような高速かつ容易に変換が行える手法は非常に有効である．RBP 法は疎行列格納になるメモリ使用量を従来の疎行列格納方式に比べ最大 27%以上も削減しながら，GMRES 演算性能の低下がない手法であると言える．

次に疎行列の CPU メモリから GPU メモリへの転送時間を比較すると，CSR，ELL,

ELL-R に比べ，RBP-CSR，RBP-ELL，RBP-ELL-R のデータ転送時間が平均で，15.1%, 10.6%, 10.4%短縮された．また最大のデータ転送時間削減率となったのは pwtk であり，RBP-CSR では 27.6%, RBP-ELL では 26.1%, RBP-ELL-R では 27.5%，既存疎行列格納方式に比べ，データ転送時間を短縮した．PatComp のデータ転送時間は CSR のデータ転送時間と比べ，平均で 19.2%短くなっている．pwtk を用いた際，最も短縮した時間が大きく，31.9%のデータ転送時間短縮に成功した．この結果から RBP 法，PatComp によるメモリ使用量削減に伴いデータ転送時間も削減されたことを確認した．しかしメモリ使用量削減が困難であった parabolic_fem，thermal2 においてはデータ転送時間の増加が確認された．

RBP-ELL，RBP-ELL-R では SpMV 演算時間の増大が見られた疎行列において，GMRES 演算時間も増大した．nd24k では ELL で 32.4%，ELL-R で 44.6%の演算時間が伸びている．しかしその他の GMRES 演算時間の増大が見られた疎行列では，10%以下の演算時間増加に留まっている．この結果から多くの疎行列において，RBP 法を用いても，既存の疎行列格納方式と遜色ない時間で GMRES を実行でき，使用するメモリ使用量は大幅に削減可能である．

5.9.1.2 CPU による GMRES 演算時間

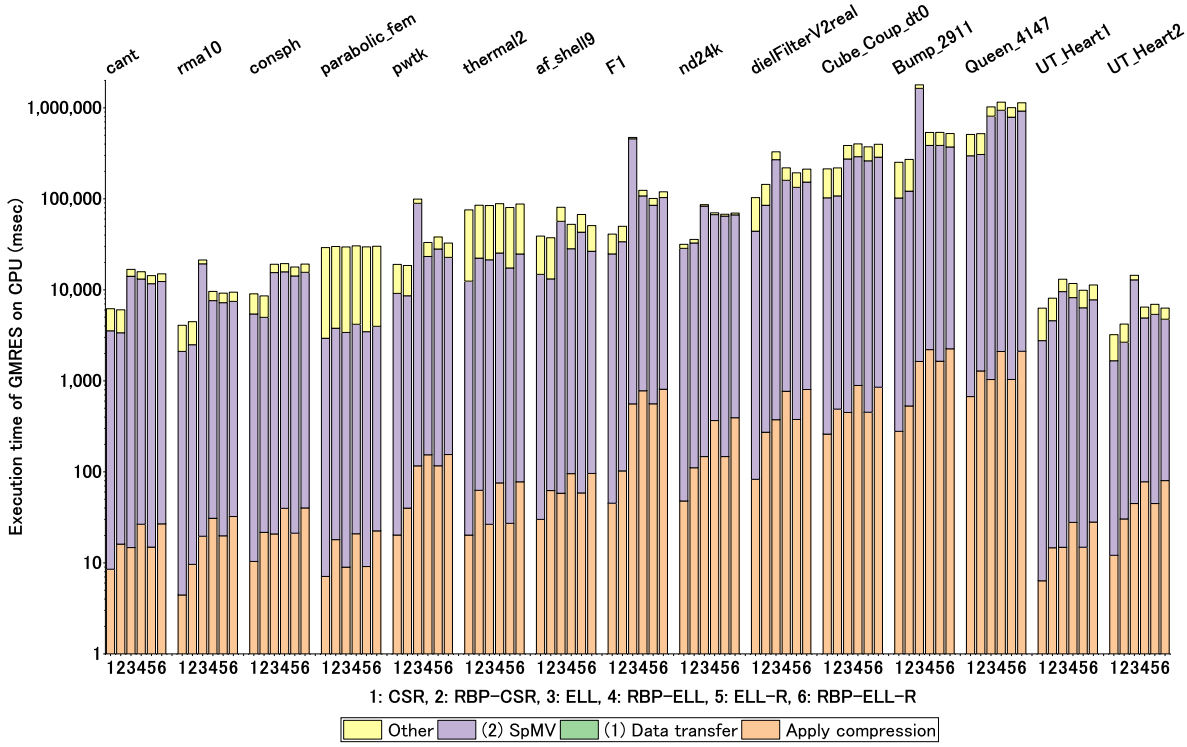

図 5.16: Execution time of GMRES on CPU

次に，GPU を用いることが反復法，そして SpMV の高速化に有効であることを確認するため，図 5.5 の赤枠内の処理を全て CPU で行った場合の処理時間を図 5.16 に示す．使用した CPU のコア数は 1 である．縦軸は対数軸となっている．図中の各演算時間の内訳や反復法の最大反復回数，収束判定の精度は 5.9.1.1 節の GPU を用いた GMRES と同じ条件である．

CPU 上で CSR，ELL，ELL-R，RBP-CSR，RBP-ELL，RBP-ELL-R を用いた GMRES 演算時間 (図 5.16 と GPU 上でのそれぞれの GMRES 演算時間 (図 5.15)) を比較すると，平均で 14.9, 122.1, 56.7, 27.0, 70.3, 67.7 倍の高速化となった．ELL は GPU のようなベクトル型マシンに適した疎行列格納方式であることから，CPU での SpMV 演算時間は非常に長いが，GPU では非常に短く，SpMV が高速であることが示されている．一方で CSR は CPU，GPU 両方で多く使用されることから，CPU を用いた場合と GPU を用いた場合の GMRES の演算時間の差は ELL ほど大きくなっていない．しかしながら，それでも GPU を用いることで約 15 倍の高速化を達成している．これらの結果から GPU を用いることが反復法，そして SpMV の高速化に対し非常に有効であることを確認した．反復法においてデータ転送は 1 回のみであるため，反復毎に実行される SpMV の演算時間が占めるの割合が大きくなる．図 5.15，図 5.16 から，CPU でボトルネックとなっていた SpMV 演算時間が GPU を用いることで大きく短縮されている．そのためデータ転送が発生したとしても，GPU を GMRES の高速化に使用することの利益は大きい．

また文献 [13] において，8GPU で構成された GPU クラスタを用いることで，12CPU で構成された CPU クラスタを用いた場合に比べ，GMRES 演算速度を約 8 倍高速化することに成功している．この結果からも GPU の反復法高速化に対する有効性が示されている．

5.9.2 COO 形式から各疎行列格納方式への変換時間

図 5.5 中の緑色の枠で示した COO 形式から各疎行列格納方式への変換時間の測定結果を，図 5.17 に示す．COO から各疎行列格納方式への変換は CPU 上で行われる．CSR, ELL，ELL-R，PatComp の変換時間は，COO からそれぞれの疎行列格納方式への変換に要した時間である．RBP-CSR，RBP-ELL，RBP-ELL-R の変換時間は，一度 COO を CSR，ELL，ELL-R に変換し，その後 RBP 法を適用するために要した時間である．COO から CSR，ELL，ELL-R，RBP-CSR，RBP-ELL，RBP-ELL-R，PatComp への変換プログラムは C 言語により記述されている．

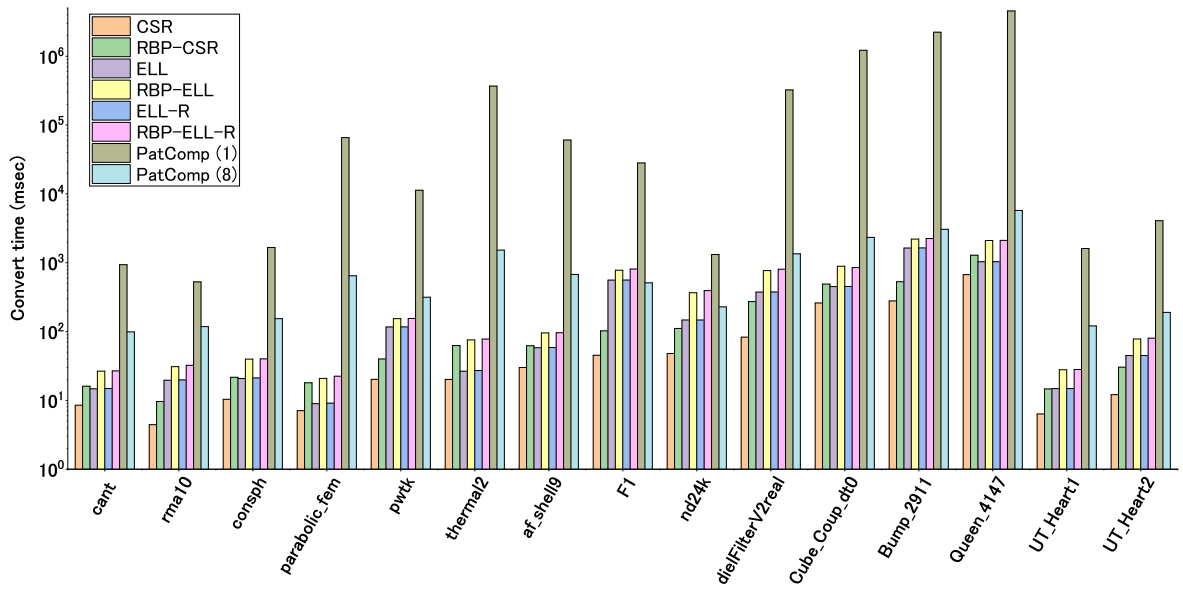

図 5.17: Convert time of each formats

RBP 法は既存の疎行列格納方式に対し圧縮を行う手法である．そのため図 5.17 では CSR，ELL，ELL-R に比べ，RBP-CSR，RBP-ELL，RBP-ELL-R の変換時間は長くなっている．既存の疎行列格納方式の変換時間から，RBP 法を適用した疎行列格納方式の変換時間を差し引いた時間が，RBP 法のオーバヘッドである

しかしながら，既存の疎行列格納方式への RBP 法の適用時間を含めても，RBP 法を用いた疎行列格納方式を使用した GMRES の演算時間は，既存の疎行列格納方式を用いた演算時間と遜色ない．

RBP-CSR と PatComp(1) への変換時間を比較すると，RBP-CSR の方が平均で 1529 倍の高速化を達成している．また RBP-CSR と並列化を行った PatComp(8) の変換時間を比較すると，RBP-CSR の方が平均で 10 倍高速な結果となった．RBP-ELL や RBP-ELL-R と PatComp への変換時間を比較しても，RBP 法の変換は非常に高速な結果となっている．よって RBP 法が目的として掲げていたメモリ使用量を削減しながらも高速な変換手法を達成していることを確認した．

5.10 RBP 法の適用可能な疎行列

RBP 法は CSR と比較しメモリ使用量が大幅に少なくなっており，PatComp と比べ変換時間が非常に短い手法である．そのため PatComp の適用が困難な変換処理がシミュレーションの時間ステップごとに必要な FEM モデルの形状が変化する問題でも RBP 法の適用が可能である．解析対象に破壊現象や亀裂発生等が発生し FEM モデルの形状が変化し

た場合，各時間ステップごとに非ゼロ要素の位置情報を含め更新が必要となるが，RBP 法では変換が高速であることから更新のコストが非常に少ない．

また反復法の収束までに必要な反復回数が少ない問題にも RBP 法は有効である．反復回数が少なくなることで SpMV に対し変換に要する時間が支配的になることから変換が高速な RBP 法が有効となる．本章での GMRES 評価実験では用いていないが，収束性を改善するための前処理を反復法と併用するケースが多々ある．反復法の前処理を使用することで収束性は大きく改善されることから，反復回数は少なくなる．そのような場合には変換に要する時間が支配的となることから RBP 法が活用できると考える．

次に RBP 法を用いて従来の疎行列格納方式よりメモリ使用量が削減可能な条件を検討する．まず RBP-CSR のメモリ使用量を疎行列内の連続した非ゼロ要素のブロック数を用いて表すと式（5.4) となる．ここで N_{con} は疎行列内に存在する連続した非ゼロ要素のブロック数を表す．N，N_z，N_{non} はこれまでと同様に疎行列の行数，非ゼロ要素数，不連続な非ゼロ要素数である．また値の格納には 8byte，その他の要素の格納には 4byte 必要と仮定する．

$$MemUsage_{RBP-CSR} = 4 \times 3(N+1) + 4 \times 2N_{con} + 8(N_z - N_{non}) + 4N_{non} + 8N_{non} \quad (5.4)$$

式（5.4）を用いて従来手法である CSR よりメモリ使用量を削減可能な条件を式（5.5）に示す．疎行列内の非ゼロ要素を一度参照するだけで，その疎行列が RBP 法でメモリ使用量削減可能か判別することが可能である．

$$8N_z + 4N_z + 4(n+1) > 4 \times 3(N+1) + 4 \times 2N_{con} + 8(N_z - N_{non}) + 4N_{non} + 8N_{non} \quad (5.5)$$

次に RBP-ELL のモリ使用量を疎行列内の連続した非ゼロ要素の数に着目し表すと式 (5.6) となる．ここで $N_{zperrow}$ は 1 行あたり連続した非ゼロ要素の値の数（不連続な非ゼロ要素の値を除く）．$N_{conperrow}$ は 1 行あたり連続した非ゼロ要素のブロック数を表す．$Max(N_{zperrow})$，$Max(N_{conperrow})$ は全ての行の内最も大きい $N_{zperrow}$，$N_{conperrow}$ の値を表す．

$$\begin{aligned} MemUsage_{RBP-ELL} = 8N \times Max(N_{zperrow}) + 2 \times 4N \times Max(N_{conperrow}) \\ +8N_{non} + 4N_{non} + 4(N+1) \end{aligned} \quad (5.6)$$

式（5.6）より，既存手法である ELL より RBP-ELL のメモリ使用量が少なくなる条件を式（5.7）に示す．RBP-ELL も疎行列内の連続した非ゼロ要素の数，不連続な非ゼロ要素の数を調査することでメモリ使用量削減が可能か判別することが可能である．また RBP-ELL-R も RBP-ELL と同様の条件式（5.7）により判別可能である．

$$8N \times K + 4N \times K > 8N \times Max(N_{zperrow}) + 2 \times 4N \times Max(N_{conperrow}) \\ +8N_{non} + 4N_{non} + 4(N+1) \tag{5.7}$$

PatComp と RBP 法の使い分けとして，まず式（4.2)，式（5.5）と式（5.7）を使用して対象となる疎行列をそれぞれの疎行列格納方式で格納した際のおおよそのメモリ使用量を算出し，使用する GPU のメモリ容量に合わせた選択を行う．PatComp のメモリ使用量の方が RBP-CSR のメモリ使用量より少ない傾向にあるため，PatComp を用いることで RBP 法より大規模な数値シミュレーションを扱うことが可能である．しかし PatComp では COO からの変換時間が RBP 法に比べ非常に長いことから，シミュレーションの時間ステップごとに FEM モデルの形状が変化する場合，時間ステップ毎に変換が必要となり，変換時間が非常に大きくなる．このような場合には RBP 法を選択することで変換時間がボトルネックとなることを抑制することが可能である．

5.11 おわりに

RBP 法は PatComp のボトルネックとなっていた疎行列からの変換時間を短縮するため，圧縮方法を単純化した手法である．RBP 法により変換処理を高速に行うことで，PatComp が適していない時間ステップごとに疎行列の形状が変化する問題に対してもメモリ使用量の削減を達成することが可能である．

FDM や FEM で生成される疎行列は，ある領域とその隣接する領域からの影響を反映するため，疎行列には連続した非ゼロ要素が多々出現する．従来の疎行列格納方式では連続した非ゼロ要素が存在した場合であっても，全ての非ゼロ要素の情報を格納していた．しかし SpMV は先頭と末尾の非ゼロ要素の列番号のみで実行可能である．そこで RBP 法では連続した非ゼロ要素の先頭と末尾の列番号のみを格納することで，列番号に関する情報量を減らし，メモリ使用量の削減を行う．

RBP 法と同様に連続した非ゼロ要素に着目した疎行列格納方式である BCCOO では複数行，複数列に渡る固定サイズのブロックを用いて疎行列を分割し，各ブロックごとに位置情報を与えることで，位置を示す要素数を削減している．しかし固定サイズのブロックを用いていることから 0 要素が含まれ，メモリ使用量が増加することもあるため，最適なブロックサイズを設定するために多くの変換時間を必要としていた．そこで RBP 法では変換時間を短縮するため，列番号の削減を行う対象範囲を 1 行中の連続した非ゼロ要素に限定することで，圧縮を行い，変換するために必要な時間を既存手法に比べ短くなるよう設計した．

5.4 節，5.5 節において RBP 法の CSR と ELL に対する適用方法について述べた．連続した非ゼロ要素を 1 つの Block とし，従来の非ゼロ要素 1 個と同じ扱いをすることで，従来手法である CSR，ELL へ容易に適用可能である．また不連続な非ゼロ要素に関しては，連続した非ゼロ要素とは別の領域に格納する．領域を分割することで，不連続な非ゼロ要

素の格納に必要となるメモリ使用量を削減すると共に，SpMV 演算性能を向上させる狙いがある．

RBP-CSR，RBP-ELL を使用した SpMV は，連続した非ゼロ要素の先頭の列番号を反復ごとにインクリメントすることで本来必要となる列番号を復元する．従来手法である CSR や ELL は反復毎にアクセス速度が低速な Global memory へアクセスするため，長いメモリアクセス時間を消費する．一方 RBP 法はレジスタの値をインクリメントするだけで列番号を得ることができるため，Global memory へのメモリアクセス回数削減による演算時間の短縮も期待することができる．

本章では RBP 法の趣意である既存の疎行列格納方式 CSR や ELL より少ないメモリ使用量で疎行列を格納しながらも，変換に必要となる時間が短い疎行列格納方式が達成されているか確認するため，評価実験を行った．評価実験では実際の数値シミュレーションで使用された疎行列を用いて評価を行った．

まず最初のメモリ使用量の評価実験において，RBP 法は既存手法である CSR や ELL に比べ最大 27%以上，平均で 11 %以上のメモリ使用量削減を達成した．また BCCOO と FEM で使用された疎行列で比較すると平均で 3.4%メモリ使用量が多い結果となったが，BCOOO は変換時間に多くの時間を要する．RBP 法は変換時間が BCCOO に比べ格段に変換に必要となる計算量が少ないながらもメモリ使用量は 3.4%の差であり，時間発展ごとに変換を要する問題においては RBP 法は非常に有効である．

SpMV 演算時間の評価では RBP 法を用いた RBP-CSR，RBP-ELL 共に平均で 1.5 倍以上の高速化を達成した．この結果から RBP 法は既存手法に比べて演算性能を損なうことなくメモリ使用量の削減が可能な手法であることを確認した．

また RBP 法は，変換処理が反復法のボトルネックとならないよう変換処理を単純化し高速化を測った．結果として GMRES を用いた反復法の演算時間の評価では，変換時間を含めても既存の格納方式と遜色ない演算時間で処理が完了することを確認した．また逐次処理での PatComp の変換時間と比較して 1529 倍，8 並列での変換と比較して 10 倍の高速化を達成した．これらの結果から RBP 法の目的である変換処理の高速に行いながらもメモリ使用量の削減を可能にすることを達成した．

第6章　結言

6.1　本研究の総括

本研究では大規模かつ高精度な数値シミュレーションを GPGPU を用いてオンサイト環境で実現する際，ボトルネックとなる連立一次方程式求解の省メモリ化に取り組んだ．

2 章では，数値シミュレーションと様々な解析手法の特色を述べ，連立一次方程式求解の省メモリ化への道筋を説明した．数値シミュレーションの代表的手法である FDM，FEM は，最終的に連立一次方程式求解に帰着する．GPGPU を用いて連立一次方程式の求解手法を高速化する際には，この連立一次方程式を表す巨大な疎行列を GPU のデバイスメモリへ格納する必要があるが，格納しきれない場合には大幅に演算性能が低下する．そのため，連立一次方程式を表す疎行列格納に必要となるメモリ使用量の削減が必須であることを示した．またメモリ使用量を削減するだけでなく，疎行列を使用する処理，SpMV を GPGPU 上で効率よく実行できるようなデータ構造で格納する必要がある．

3 章では，GPGPU に関して説明を行い，これまでに GPGPU 向けに提案されてきた疎行列格納方式について記述した．GPGPU は描画処理専用であった GPU を汎用的な処理に使用し，高速化を図る技術である．GPU には多くの演算コアが搭載されており，高い並列度でアプリケーションを並列化することが可能である．しかし，GPGPU を用いてアプリケーションを実行するためには，必要なデータを GPU メモリへ格納しなければいけない．例えば反復法では，巨大な疎行列を GPU メモリへ格納する必要があるが，格納しきれない場合には CPU メモリとのデータの入れ替えが頻発するため，大きなボトルネックとなる．そのため疎行列のデータ量を削減し，少ないメモリ使用量で格納できるよう，対処する必要がある．

代表的なデータ圧縮手法として，ハフマン符号化，LZ77 圧縮，LZ78 圧縮や文法圧縮手法などが存在するが，これらの手法は過去に出現した文字列の並びへのポインタを使用することでデータ量を削減する．しかし GPU を用いた高速化においては高い並列度での並列化が可能である必要がある．そのため代表的なデータ圧縮手法は疎行列の圧縮には不向きであることを説明した．通常のデータ圧縮では疎行列を少ないメモリ使用量で格納できないため，一般的に疎行列を格納する際は疎行列格納方式が使用される．疎行列格納方式では，疎行列内の非ゼロ要素をいかに少ない情報量で表すかを重視しており，1 つ 1 つの非ゼロ要素が独立して格納されるため高い並列度での SpMV 実行に使用可能である．これまでに GPGPU を用いて SpMV を高速化するために様々な疎行列格納方式が提案されてきた．しかしながら，FDM，FEM で生成される疎行列の非ゼロ要素のパターン性を考

慮した疎行列格納方式は提案されていないことを示した．

4 章では，既存手法では考慮されていない疎行列内の非ゼロ要素のパターン性を考慮した Pattern Compression (PatComp) 法を提案した．PatComp は変換処理に多くの計算が必要となることから，時間ステップ毎に亀裂や破壊による FEM モデルの格子形状が変化がなく，疎行列の形が変わらない問題を対象とした手法である．実際の数値シミュレーションで使用された疎行列を用いた予備実験において，疎行列内の非ゼロ要素の並び方にはパターン性があることを明らかにした．多くの疎行列では，各行の非ゼロ要素の要素の並びはごくわずかなパターンで表すことが可能であることが判明した．そこで PatComp 法では疎行列内のパターンを格納し，複数の行がそのパターンを読み出すことで SpMV を実行する．複数の行が同じパターンであるとき，格納する必要があるパターンは 1 つであることから，大幅なメモリ使用量の削減が可能である．

5.3 節では PatComp 法の目的が達成されているか確認するためメモリ使用量，SpMV 演算時間，GMRES 演算時間，疎行列からの変換時間を評価した．メモリ使用量の評価では，PatComp を用いることで，CSR に比べ最大 31.1%のメモリ使用量の削減を達成した．15 個中 4 個の疎行列において 30%以上，12 個の疎行列において 25%以上のメモリ使用量を削減しており，PatComp の有効性を示した．また SpMV 演算時間も既存の疎行列格納方式 CSR と遜色ない結果であることを確認した．しかしながら疎行列から PatComp への変換時間は多くの時間を要し，GMRES のボトルネックとなることを確認した．そのため数値シミュレーション中に 1 回しか変換を要しない問題には適しているが，時間ステップごとに変換が必要となるた問題には適していない．

5.3 章において，PatComp の欠点である変換時間の高速化を図るため，Row Block Packing(RBP) 法を提案した．RBP 法はメモリ使用量を削減しながらも，高速に変換を行える手法を目的としている．高速な変換を行えるようにすることで，解析対象に亀裂や破損が発生し，疎行列の形状も時間ステップごとに変化する問題に対してもメモリ使用量の削減が行えるようになる．RBP 法は FDM や FEM で生成される疎行列内の非ゼロ要素の連続性に着目した疎行列圧縮手法である．RBP 法は連続する非ゼロ要素のうち，先頭と末尾の要素の列番号のみを格納することで，列番号格納に必要となるメモリ領域を削減する．PatComp と異なり，テーブルを用意する必要がなく，非ゼロ要素の列番号を順に確認して行き，連続している場合のみ先頭と末尾の列番号を格納するため，非ゼロ要素数に対して線形の計算量となる．CSR と ELL に対する RBP 法の適用フローを示し，その後 RBP-CSR，RBP-ELL を用いた SpMV カーネルを示した．RBP 法を用いることで，低速な Global Memory へのアクセス回数を削減することが可能である．

5.6 節では RBP 法の目的が達成されているか評価実験を行った．まずメモリ使用量の評価実験では，RBP 法を用いることで CSR のメモリ使用量を最大 27.9%，ELL と ELL-R のメモリ使用量を最大 28.0%削減することに成功した．SpMV 演算時間の評価では，既存の疎行列格納方式を用いた SpMV と RBP 法を用いた SpMV の演算性能がほぼ同等であることを示した．RBP-CSR は CSR に比べ平均 1.82 倍，RBP-ELL は ELL に比べ平均 1.57 倍，RBP-ELL-R は ELL-R と比べ平均 1.14 倍であった．GMRES を用いた評価実験では，各疎

行列格納方式への変換時間を含めたGMRES演算時間を評価した．その結果，RBP法を適用する時間を含めてもRBP-CSRで平均3.1%，RBP-ELLで平均10.4%，RBP-ELL-Rで平均1.4%，既存の格納方式を用いたGMRES演算時間より短縮した．この結果から，提案したRBP法は既存の疎行列格納方式の演算性能はそのままに，メモリ使用量を大幅に削減可能であることを示した．また変換も高速であることから，変換時間がGMRESのボトルネックとならないことを確認した．PatCompの変換時間とRBP-CSRへの変換時間を比較すると10倍以上RBP-CSRへの変換の方が高速であった．よってRBPの目的であった変換時間の短縮が達成されたことを確認した．

本研究で提案したPatComp法，RBP法を活用し，数値シミュレーションで使用するメモリ使用量を削減することで，さらに大規模かつ高精度な数値シミュレーションを同じ環境で行うことが可能となる．医療分野ではさらに大規模かつ高精度な数値シミュレーションがオンサイト環境で可能になり，検診精度の向上，検診時間の短縮など患者にとって，様々な利益がある．またオンサイト環境だけでなく大規模な演算環境においてもメモリ使用量の削減は有効である．地震発生時など緊急の場合に大規模なシミュレーションを行う際，RBP法やPatComp法を用いることで，解析可能な範囲を広げるができ，被害抑制に貢献することが予測される．

6.2 今後の展望

本研究はメモリ使用量の削減が第一目的であったため，RBP法，PatComp法のGPUに対する最適化は行わなかった．しかしRBP法，PatComp法のSpMVカーネルを最適化することでさらに演算性能向上が期待できる．そのためRBP法，PatComp法のメモリ使用量をそのままにSpMVの演算性能を高める手法についても今後研究に取り組みたいと考えている．

また本研究では，疎行列と密ベクトルの積（SpMV）を対象として疎行列格納方式を提案，評価を行った．しかし疎行列を使用する処理は，疎行列と疎行列，疎行列と疎ベクトルとの積など他にも多く存在する．そのため今後提案した疎行列格納方式がこれらの処理に適用した際にどの程度有効なのか評価を行う．そしてそれぞれの疎行列を使用する処理に適した疎行列の格納方式となるようRBP法，PatComp法を改良する．

謝辞

本学に入学以降，博士前期課程，博士後期課程と長きにわたり，本学の井口　寧教授には手厚いご指導を受け賜りました．こうして今学位論文を書き終えることができましたのも，井口 寧教授の暖かい励まし言葉のおかげです．研究に対するご指導のみならず，学生生活にも有益なアドバイスを頂き，大変充実した学生生活を送ることができました．この場を借りて深く感謝するとともにお礼申し上げます．

学位論文の審査を引き受けていただきました本学の金子 峰雄教授，田中 清史准教授，本郷 研太准教授，富山高等専門学校の古山 彰一教授に深く感謝いたします．古山 彰一教授には，富山高等専門学校時代に研究をご指導いただき，研究者を志すきっかけを与えていただきました．心より感謝申し上げます．

また共同研究において多くのご協力を頂きました富士通株式会社の渡邉 正宏様，岩村 尚様，米田 一徳様にお礼申し上げます．お忙しい中貴重な時間をいただき，研究に対し多くのご意見をいただきました．心から感謝申し上げます．

研究室のゼミで，多くの御意見，ご指導をいただいた河野 隆太助教をはじめとする，井口研究室のメンバーに感謝と御礼を申し上げます．井口研究室の OB である荒木光一様には，研究だけでなく，私生活においても支えていただきました．深く感謝とお礼を申し上げます．

最後に，日頃より暖かく見守っていただいた妻の知実，両親，親族に心から感謝いたします．

参考文献

[1] Min, J.K., Taylor, C.A., Achenbach, S.K., Bon Kwon Leipsic, J., Nørgaard, B.L., Pijls, N.J., De Bruyne, B. : Noninvasive Fractional Flow Reserve Derived From Coronary CT Angiography Clinical Data and Scientific Principles, *JACC: Cardiovascular Imaging*, Vol.8, No.10, pp.1209-1222 (2015).

[2] Modi, B. N., Sankaran, S., Kim, H. J., Ellis, H., Rogers, C., Taylor, C. A., Rajani, R., Perera, D.:Predicting the physiological effect of revascularization in serially diseased coronary arteries: Clinical validation of a novel CT coronary angiography-based technique, *Circ Cardiovasc Interv.*, Vol.12, No.2, pp.1-8 (2019).

[3] Kato, M., Hirohata, K., Kano, A., Higashi, S., Goryu, A., Hongo, T., Kaminaga, S., Fujisawa, Y. :Fast CT-FFR Analysis Method for the Coronary Artery Based on 4D-CT Image Analysis and Structural and Fluid Analysis, Proc. *International Mechanical Engineering Congress and Exposition* (IMECE2015), ASME, p.1-10 (2015).

[4] Coenen, A., Lubbers, M.M., Kurata, A., Kono, A., Dedic, A., Chelu, R.G., Dijkshoorn, M.L., Gijsen, F.J., Ouhlous, M., van Geuns, .M., Nieman, K. : Fractional Flow Reserve Computed from Noninvasive CT Angiography Data: Diagnostic Performance of an On-Site Clinician-operated Computational Fluid Dynamics Algorithm, *Radiol.*, Vol.274, No.3, pp.674-683 (2015).

[5] B. He, H. P. Huynh and R. G. S. Mong, "GPGPU for real-time data analytics," 2012 IEEE 18th International Conference on Parallel and Distributed Systems, Singapore, 2012, pp. 945-946, doi: 10.1109/ICPADS.2012.156.

[6] R. Takahashi and U. Inoue, "Parallel Text Matching Using GPGPU," 2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing, Kyoto, 2012, pp. 242-246, doi: 10.1109/SNPD.2012.28.

[7] W. Blewitt, G. Ushaw and G. Morgan, "Applicability of GPGPU Computing to Real-Time AI Solutions in Games," in IEEE Transactions on Computational Intelligence and AI in Games, vol. 5, no. 3, pp. 265-275, Sept. 2013, doi: 10.1109/TCIAIG.2013.2258156.

[8] Sugiura, S., Washio, T., Hatano, A., Okada, J., Watanabe, H., Hisada, T.: Multi-scale simulations of cardiac electrophysiology and mechanics using the University of Tokyo heart simulator, Progress in Biophysics and Molecular Biology, Vol.110, pp.380-389(2012).

[9] P. Agarwal and S. L. Girshick, "Numerical Simulations of Nanodusty RF Plasmas," in IEEE Transactions on Plasma Science, vol. 39, no. 11, pp. 2760-2761, Nov. 2011, doi: 10.1109/TPS.2011.2157945.

[10] W. Ohira, K. Honda and M. Nagai, "Tsunami inundation damping performance of mangrove based on two-dimensional numerical simulation," 2015 IEEE International Geoscience and Remote Sensing Symposium (IGARSS), Milan, 2015, pp. 2727-2730, doi: 10.1109/IGARSS.2015.7326377.

[11] J. Taniguchi, M. Miyoshi, T. Baba and H. Aki, "A numerical simulation of drainage of influent water caused by Tsunami at Kawauchi, Tokushima," 2016 Techno-Ocean (Techno-Ocean), Kobe, 2016, pp. 489-492, doi: 10.1109/Techno-Ocean.2016.7890703.

[12] Yousef, Saad.: Iterative methods for sparse linear systems, second edition, Society for Industrial and Applied Mathematics, 2003.

[13] Jacques, M.Bahi., Raphaël, Couturier., and Lilia, Ziane Khodja.: Parallel GMRES implementation for solving sparse linear systems on GPU clusters, HPC '11 Proceedings of the 19th High Performance Computing Symposia, pp.12-19, Boston, USA, April 2011.

[14] Chong, Chen., Tarek, M.Taha.: A comunication reduction approach to iteratively solve large sparse linear systems on a GPGPU cluster, Springer US, Journal of Cluster Computing, vol.17, no.2, pp.327-337, June 2013.

[15] Byron, DeVries., Joe, Iannelli., Christian, Trefftz., Kurt, A, O'Hearn. and Greg, Wolffe.: Parallel implementations of FGMRES for solving large sparse non-symmetric linear systems, 2013 International Conference on Computational Science, vol.18, pp.491-500, USA (2013).

[16] Nathan Bell and Michael Garland.: Implementing sparse matrix-vector multiplication on throughput-oriented processors, In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC '09). Association for Computing Machinery, New York, NY, USA, Article 18, 1–11. DOI:https://doi.org/10.1145/1654059.1654078

[17] Dahai Guo and William Gropp. Adaptive thread distributions for SpMV on a GPU. In Proceedings of the Extreme Scaling Workshop (BW-XSEDE ' 12). University of Illinois at Urbana-Champaign, USA, Article 2, 1–5.

[18] Jeff Bolz, Ian Farmer, Eitan Grinspun, and Peter Schröder.: Sparse matrix solvers on the GPU: conjugate gradients and multigrid, In ACM SIGGRAPH 2003 Papers (SIGGRAPH ' 03). Association for Computing Machinery, New York, NY, USA, 917–924. DOI:https://doi.org/10.1145/1201775.882364

[19] Bor-Yiing Su and Kurt Keutzer.: ClSpMV: A Cross-Platform OpenCL SpMV Framework on GPUs, In Proceedings of the 26th ACM international conference on Supercomputing (ICS ' 12). Association for Computing Machinery, New York, NY, USA, 353–364. DOI:https://doi.org/10.1145/2304576.2304624

[20] Salvatore, Filippone., Valeria, Cardellini., Davidem Barbieri., et al. : Sparse Matrix-Vector Multiplication on GPGPUs, ACM Transactions on Mathematical Software (TOMS), vol.43 Issue 4, March 2017

[21] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," in Proceedings of the IRE, vol. 40, no. 9, pp. 1098-1101, Sept. 1952. doi: 10.1109/JRPROC.1952.273898

[22] G. G. Langdon: An Introduction to arithmetic coding, IBM Journal of Research and Development, vol. 28, no. 2, pp.135-149 1984

[23] A. H. Robinson and C. Cherry, "Results of a prototype television bandwidth compression scheme," in Proceedings of the IEEE, vol. 55, no. 3, pp. 356-364, March 1967. doi: 10.1109/PROC.1967.5493

[24] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," in IEEE Transactions on Information Theory, vol. 23, no. 3, pp. 337-343, May 1977. doi: 10.1109/TIT.1977.1055714

[25] J. A. Storer and T. G. Szymanski, "Data compression via textual substitu-tion", J. ACM, vol. 29, no. 4, pp. 928-951, 1982.

[26] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," in IEEE Transactions on Information Theory, vol. 24, no. 5, pp. 530-536, September 1978. doi: 10.1109/TIT.1978.1055934

[27] Katz, Phillip W.: "String Searcher, and Compressor Using Same", US 5051745 A, URL:https://lens.org/176-808-555-607-469, 1991

[28] Welch, "A Technique for High-Performance Data Compression," in Computer, vol. 17, no. 6, pp. 8-19, June 1984. doi: 10.1109/MC.1984.1659158

[29] A.R.Alameldeen and D.A.Wood, Frequent Pattern Compression: A Significance-Based Compression Scheme for L2 Caches, In Technical Report 1500, Computer Sciences Department, Univ. of Wisconsin-Madison, April 2004.

[30] Y.Zhang, J.Yang, and R.Gupta, Frequent Value Locality and Value-Centric Data Cache Design, In Proc. of 9th ASPLOS, pages 150–159, November 2000.

[31] N. J. Larsson and A. Moffat, "Off-line dictionary-based compression," in Proceedings of the IEEE, vol. 88, no. 11, pp. 1722-1732, Nov. 2000. doi: 10.1109/5.892708

[32] David, R. Kincaid., Thomas, C. Oppe., and David, M. Young., : IT-PACKV 2D User ' s Guide, CNA-232",
https://www.ma.utexas.edu/CNA/ITPACK/manuals/
userv2d/node3.html January 2018 Available

[33] Nathan, Bell., Michael, Garland.: Efficient sparse matrix-vector multiplication on CUDA, NVIDIA Technical Report NVR-2008-004, NVIDIA, http://www.nvidia.com/docs/IO/66889/nvr-2008-004.pdf, December 2008

[34] F.Vázquez, J. J. Fernández, E. M. Garzón, :A new approach for sparse matrix vector product on NVIDIA GPUs, Concurrency and Computation Practice and Experience, vol.23, no.8, pp.815-826, 2011.

[35] Alexander, Monakov., Anton, Lokhmotov., and Arutyun, Avetisyan., :Automatically tuning sparse matrix-vector multiplication for GPU architectures, 5th International Conference, HiPEAC 2010, Pisa, Italy, pp. 111-125, January 2010.

[36] Jee, W. Choi., Amik, Singh., Richard, W. Vuduc., :Model-driven autotuning of sparse matrix-vector multiply on GPUs, Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, India, Pages 115-126, 2010

[37] Ali, Pinar. and Michael, T. Heath.: Improving performance of sparse matrix-vector multiplication, SC' 99 Proceedings of the 1999 ACM/IEEE conference on Article No. 30, USA, November 1999

[38] M. Kreutzer et al. A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix- Vector Multiplication on Modern Processors with Wide SIMD Units. SIAM Journal on Scientific Computing, 36(5):C401–C423, 2014.

[39] M. Maggioni and T. Berger-Wolf. CoAdELL: Adaptivity and Compression for Improving SparseMatrix-Vector Multiplication on GPUs. In 2014 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pages 933–940. IEEE, 2014.

[40] M. Maggioni and T. Berger-Wolf, "AdELL: An Adaptive Warp-Balancing ELL Format for Efficient Sparse Matrix-Vector Multiplication on GPUs," 2013 42nd International Conference on Parallel Processing, Lyon, 2013, pp. 11-20. doi: 10.1109/ICPP.2013.10

[41] S. Yan et al. yaSpMV: Yet Another SpMV Framework on GPUs. In Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP ' 14, 2014.

[42] Yusuke Nagasaka, Akira Nukada, Satoshi Matsuoka: Adaptive Multi-level Blocking Optimization for Sparse Matrix Vector Multiplication on GPU, Procedia Computer Science, Vol. 80, pp. 131-142 (2016).

[43] Tomoki, Kawamura., Kazunori, Yoneda., Takashi, Yamazaki., Takashi, Iwamura., Masahiro, Watanabe., and Yasushi, Inoguchi.: A compression method for storage formats of sparse matrix in solving the large-scale linear systems, International Parallel and Distributed Processing Symposium Workshops (IPDPSW), APDCM, Lake Buena Vista, FL, USA, pp.924-931. May-June 2017.

[44] Timothy, A. Davis. and Yifan, Hu.: 2011. The university of Florida sparse matrix collection. ACM Trans. Math. Softw. 38, 1, Article 1 (December 2011), 25 pages. DOI: https://doi.org/10.1145/2049662.2049663

研究業績

- 論文誌

  1. 河村 知記，米田 一徳，岩村 尚，渡邉 正宏，井口 寧，“オンサイトでの高精度数値シミュレーション実施のためのGPU向き疎行列圧縮スキーム”，情報処理学会論文誌数理モデル化と応用（TOM），Vol. 13, No. 2, pp. 93-106 (2020).

- 国際会議

  1. Tomoki Kawamura, Yoneda Kazunori, Takashi Yamazaki, Takashi Iwamura, Masahiro Watanabe and Yasushi Inoguchi, “A compression method for storage formats of sparse matrix in solving the large scale linear systems”, In 19 the Workshop on Advances in Parallel and Distributed Computational Models (APDCM) held in conjunction with 31rd IEEE International Parallel and Distributed Processing Symposium (IPDPS), Pages 924-931, DOI: 10.1109/IPDPSW.2017.174, Mar. 2017 (査読付き)

- 国内学会

  1. 河村 知記，米田 一徳，岩村 尚，渡邉 正宏，井口 寧，“オンサイトでの高精度数値シミュレーション実施のためのGPU向き疎行列圧縮スキーム”，情報処理学会，数理モデル化と問題解決（MPS），Vol. 2020-MPS-127，No. 5，pp.1-7，Feb. 2020
  2. 河村 知記，米田 一徳，山崎 崇史，岩村 尚，渡邉 正宏，井口 寧，“GPGPU向け省メモリ指向行列格納方式の提案とGMRESへの適用”，情処学研報 2016-HPC-155, Vol. 2016-HPC-155, No. 42, pp.1-8, 長野県松本市，Aug. 2016
  3. 河村 知記，佐藤 幸紀，井口 寧，“GPU分散コンピューティングのためのデータ転送時間を考慮したデータ分割”，2014年度 電気関係学会 北陸支部連合大会，F1-2, 1 page in CD-ROM, 富山高等専門学校，Sep 2014 (平成26年度　電子通信情報学会　学生優秀論文発表賞　受賞)

- 競争的資金獲得状況

  1. 電気通信普及財団　海外渡航旅費援助　平成29年4月
  2. NEC C&C財団　国際会議論文発表者助成　平成29年4月　（採択後，辞退）