Title	GPUにおける疎行列密ベクトル積の高速化のための非ゼロ要素位置辞書圧縮を適用した疎行列格納形式の提案
Author(s)	村上, 舜
Citation	
Issue Date	2024-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/18899
Rights	
Description	Supervisor: 井口 寧, 先端科学技術研究科, 修士(情報科学)

修士論文

GPU における疎行列密ベクトル積の高速化のための非ゼロ要素位置辞書圧縮を適用した疎行列格納形式の提案

村上 舜

主指導教員 井口 寧

北陸先端科学技術大学院大学
先端科学技術研究科
（情報科学）

2023 年 3 月

Abstract

近年，数値シミュレーションの複雑化および大規模化に伴い，数百万行を超える規模の係数行列の連立一次方程式を高速に求解することが求められている．有限要素法などで離散化された偏微分方程式やグラフ解析は，行列の要素の多くが 0 である疎行列を係数行列とした連立一次方程式として表わされる．その求解には直接解法と反復解法が用いられる．しかし，直接解法はその計算量の多さや疎行列に対して完全 LU 分解をする場合，大量の fill-in が発生しメモリ使用量が増大する．そのため，大規模かつ疎な係数行列からなる連立一次方程式を求解する際には，係数行列の変形が伴わない反復解法が用いられている．

反復解法を高速化するにあたり，主要な計算時間を占める疎行列密ベクトル積（Sparse Matrix Vector products：SpMV）を高速化することが求められている．SpMV は行列の各要素に対し 1 回の積和演算のみのメモリ律速な計算である．そのため，CPU と比較して高速なメモリ帯域をもつ GPU（Graphics Processing Unit）を活用することにより高速化が図られてきた．

大規模な疎行列を，GPU の少ないデバイスメモリへ格納するにあたり，メモリ容量効率のよい CSR（Compressed Sparse Row）形式が多く用られている．しかし，CSR 形式での SpMV はストライドアクセスと reduction が必要である．そのため，メモリアクセスパターンを改善し高速に SpMV の計算が可能な SlicedELL 形式や SELL-C-σ 形式 [18] といった疎行列格納形式が提案されている．これらの格納形式では，ストライドアクセスによって発生するメモリアクセスペナルティを減らしデバイスメモリ帯域を引き出しているが，さらなる SpMV の高速化のためにはメモリアクセス回数そのものを減らす必要がある．

メモリアクセスを減らすため，非ゼロ要素の値そのものを圧縮する手法と，非ゼロ要素位置を圧縮する手法があげられる．両者を比較すると，非ゼロ要素の値を圧縮する場合，対象とする疎行列の値が実数やバイナリ，複素数など行列依存の側面が強い．非ゼロ要素位置の圧縮では整数かつ行番号と列番号の組み合わせがユニークなため，値そのものより圧縮しやすい．上記の理由により，本研究では非ゼロ要素位置の圧縮を対象とする．

非ゼロ要素位置を圧縮する手法として差分符号化と辞書圧縮があげられる．差分符号化としては CoAdELL が提案されており，行ベクトルの列番号の差分をとり，ビット数を減らすことによって容量を削減している．しかし，その差分の大きさによってはビット数を減らせず，2 の乗数 bit ではない可変長符号は GPU で計算しにくいという欠点が存在している．一方辞書圧縮方式では対象とする疎行列のパターン性によっては非常に高い圧縮率が期待できる．しかし，GPU は数スレッドを Warp と呼ばれるグループにまとめて，Warp 内の各スレッドが同一の命令を実行する．そのため辞書の単語長が異なると Warp 内のスレッドが実行している命令が異なってしまい，Warp ダイバージェンスと呼ばれる実行効率の低下が発生する．

そのため本論文では，非ゼロ要素位置情報へ辞書圧縮を適用し，メモリへのア

クセスを減らすことによって， GPU 上で SpMV を高速に計算可能とする圧縮疎行列格納形式を提案する．提案手法によって，CSR 形式と比較して最大 29.5%のメモリ使用量を削減し，SpMV の計算時間では最大 19.6%の高速化が得られた．

加えて，これらの SpMV 計算に特化した疎行列格納形式は非ゼロ要素の追加・削除が容易ではないため，より編集が容易な COO 形式で疎行列を生成・保存することが一般的である．そのため，CSR 形式や SELL-C-σ，CoD-SELL を利用するためには格納形式の変換が必要である．特に，CoD-SELL は辞書圧縮のための計算が必要であり，変換時間が大きくなると予測される．そのため，COO 形式から CSR 形式への変換と似た計算時間で変換可能な，CoD-SELL の高速な形式変換を提案する．

目 次

図 目 次

表 目 次

第1章　緒言

1.1　研究の背景

近年，物理実験や製品開発の大型化や複雑化に伴う高コスト化によって，数値シミュレーションによる模擬的実験の重要性が大きくなっている．物理現象のシミュレーションを行うためには，その自然現象を表す偏微分方程式の解を求める必要があるが，偏微分方程式の厳密解を求めることは非常に困難である．そのため，有限要素法 (FEM) や有限体積法 (FVM) を用いて偏微分方程式を連立一次方程式へ帰着させることにより，コンピュータ上でその解を求めることによって数値シミュレーションを行っている．このとき生成される係数行列はその行列サイズが非常に大きいがその要素の値の多くが 0 である疎行列となるため，0 以外の値のみを記憶することにより，必要なメモリ容量が大幅に減少する．また，グラフ理論における隣接行列を用いた最適化問題も同様に連立一次方程式の求解に帰結されるが，この時の隣接行列も疎行列となる．

このような連立一次方程式の求解法として，直接解法と反復解法が存在しているが，直接解法は係数行列の変形が必要なため，係数行列の疎性を活用することができない．そのため，疎行列を係数行列にもつ連立一次方程式の求解には，反復解法が用いられている．反復解法では，解を厳密解へ近づけるように更新していくことによって求解を行う．そのため，係数行列を疎なまま利用することが可能である．

この反復解法を実行する上で，疎行列密ベクトル積 (SpMV) が主要な計算である．そのため，疎行列を表現するための疎行列格納形式に SpMV を効率よく計算可能であることが求められている．加えて，SpMV はメモリ律速な計算であるため，CPU と比較して広いメモリ帯域幅を持つ GPU の利用がされており，GPU 上での SpMV 計算に適した疎行列格納形式であることも求められている．

1.2　目的

様々な規模および非ゼロ要素パターンの疎行列において，GPU 上で SpMV を高速に計算を行うことが目的である．SpMV 計算は疎行列のメモリ読み込みが主な

処理であるメモリ律速なアプリケーションである．その疎行列を表現するため様々な疎行列格納形式が提案されており，既存の格納形式では非ゼロ要素の値と位置情報のみを記憶することにより，密行列として格納する場合と比較して非常に少ないメモリ使用量で疎行列を表現できる．しかし，疎行列の各非ゼロ要素をメモリから読み込むためには，値に加えて位置情報もメモリから読み込む必要がある．
そのため本論文では，非ゼロ要素の位置情報に対し辞書圧縮を適用した疎行列格納形式を提案する．それにより，SpMV 計算時の SELL-C-σ 形式のメモリアクセスペナルティの少なさを維持したままメモリアクセス回数を減らすことによって SpMV 計算の高速化を行う．また，提案した疎行列格納形式の高速な形式変換アルゴリズムについても提案・評価し，疎行列格納形式の利用に必要なオーバーヘッドの削減も行う．

提案した疎行列格納形式による SpMV 計算の高速化によって，連立一次方程式の反復解法による求解の高速化が期待される．

1.3 本論文の構成

本論文は第 2 章で物理シミュレーションにおける連立一次方程式の求解および SpMV 計算の要旨を説明する．また，関連研究として疎行列格納形式の概要を提示し，それを踏まえて解決すべき課題を議論する．第 3 章で疎行列密ベクトル積の高速化のための非ゼロ要素位置辞書圧縮を適用した疎行列格納形式の提案を行う．加えて，提案した格納形式の形式変換の分析を行い，その変換時間の削減を行うことを目的に形式変換アルゴリズムの提案を行う．
提案した疎行列格納形式について第 4 章にて，理想的な非ゼロ要素パターンを持つ疎行列を用いて SpMV 計算時間の高速化とメモリ使用量の削減について評価を行い，第 3 章で示した優位性を実験により示す．加えて，提案した疎行列格納形式，CSR 形式および SELL-C-σ の SpMV 計算時間，変換時間およびメモリ容量効率の測定を行い他の疎行列格納形式と比較を行うことにより，提案形式の相対的な評価を行う．また，提案した疎行列格納形式の形式変換の並列化アルゴリズムの提案を第 5 章で行い，GPU 上での形式変換時間の測定・評価を行う．
最後に第 6 章で本研究の成果および，第 3 章，4 章および 5 章で得られた結果および考察から得られた今後の課題をまとめる．

第2章　研究の背景と関連研究

2.1　はじめに

本章では，まず数値シミュレーションが連立一次方程式として記述されており，連立一次方程式の求解手法としての直接解法と反復解法の説明を行う．また，2.2.2 章にて大規模な疎行列を係数行列として持つ連立一次方程式の求解では反復解法が適していることを述べ，2.2.3 章ではその反復解法において主要な計算である疎行列密ベクトル積 (SpMV) の説明を行う．2.3 章では SpMV の計算の高速化に利用されているハードウェアである GPU の基本的な説明を述べる．加えて，その GPU 上で SpMV を高速に計算するための疎行列格納形式の先行研究を 2.4 章で述べ，各格納形式の問題点の分析を行う．

2.2　数値シミュレーションにおける連立一次方程式とその求解方法

さまざまな物理現象のシミュレーションを行うためには，その自然現象を表す偏微分方程式の解を求める必要がある．しかし，偏微分方程式の厳密解を求めることは非常に困難であるため，数値計算で求解するため，有限要素法 (FEM) や有限体積法 (FVM) を用いて連立一次方程式へ帰着させることにより，コンピュータ上でその解を求めることが一般的である．このとき生成される係数行列はその行列サイズが非常に大きいがその要素の値の多くが 0 である疎行列となる，また，グラフ理論における隣接行列を用いた最適化問題も同様に連立一次方程式の求解に帰結されるが，この時の隣接行列も疎行列となる．そのため，多くの科学技術計算において，疎行列を係数行列にもつ連立一次方程式を高速に求解することが求められている [1].

2.2.1　直接解法

連立一次方程式を安定的に求解する手法として直接解法があげられる．ガウスの消去法に代表される直接解法は，係数行列を変形し，変数を順に消去していくことで元の方程式を一元一次方程式に帰着させていく方法である．直接解法は有

限回の操作で安定して解が得られる利点がある．しかし，係数行列の変形が必要であり，計算量が行数の三乗であるため，大規模かつ疎な係数行列を持つ連立一次方程式を求解ことは非常に大きなメモリ容量および計算時間が必要となる．

2.2.2 反復解法

一方，反復解法は，係数行列と現在の解ベクトルの積をを求めた後，右辺ベクトルと比較し解を厳密解へ更新するとによって求解を行う．直接解法と異なり，係数行列の変形を伴わないため，係数行列の疎性をそのまま利用することができる．また，1 回の反復計算の計算量が直接解法と比較して非常に少ない．そのため，大規模な疎行列を係数行列にもつ連立一次方程式の求解には，反復解法が用いられることが多い．

反復解法は係数行列の対称性や条件数によって適切な手法が異なり，代表的な手法として CG 法や GMRES 法が挙げられる．それぞれ更新式は異なるが，必要となる主な計算は，疎行列密ベクトル積，ベクトルの定数倍および加算，ベクトル同士の内積である．そのうち最も計算量を要するのは疎行列密ベクトル積であり，反復解法において疎行列密ベクトル積はその求解時間を決める重要な計算である．

2.2.3 疎行列密ベクトル積 (Sparse Matrix-Vector Multiplication：SpMV)

疎行列密ベクトル積は，疎行列 $A \in \mathbb{R}^{n \times m}$ および密ベクトル $x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_m \end{pmatrix} \in \mathbb{R}^m$ に対して

$$Ax = y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix} = \begin{pmatrix} a_{11} \times x_1 + a_{12} \times x_2 \ldots a_{1m} \times x_m \\ a_{21} \times x_1 + a_{22} \times x_2 \ldots a_{2m} \times x_m \\ \vdots \\ a_{n1} \times x_1 + a_{n2} \times x_2 \ldots a_{nm} \times x_m \end{pmatrix} \tag{2.1}$$

として表される計算である．SpMV は疎行列 A の要素あたり 1 回の積和演算しか行わないメモリ律速な計算である．そのため，CPU と比較して高速なメモリ帯域を持つ GPU の利用がされている [2][3]．2.2.2 章で述べた通り反復解法において反復のたびに実行される主要な計算であり，SpMV を GPU で高速に計算することが求められている [4]．そのため，SpMV 計算を高速に行うための疎行列格納形式が

多く提案されている．[5][6]

2.3　GPGPU

GPU はゲームなどの描画処理を高速に行うプロセッサとして発展してきたが，並列度の高い大量のデータを高速に処理可能という点および，半導体製造技術のの微細化が進んだことによって浮動小数点演算を実行可能になった．そのため近年，Graphics Processing Unit(GPU) を科学計算のアクセラレータとして利用する General-Purpose computing on Graphics Processing Units(GPGPU) が注目されている．

表 2.1 へ GPU である NVIDIA A100 80GB PCIe および NVIDIA H100 PCIe と CPU である AMD EPYC 7302 の理論性能を示す．また，図 2.1 へ NVIDIA A100 のアーキテクチャの全体を示す．

表 2.1: NVIDIA A100 80GB PCIe と AMD EPYC 7302 の理論性能 [7]，[8]，[9]

	A100 80GB PCIe	H100 PCIe	AMD EPYC 7302
倍精度浮動小数性能	9.7 TFLOPS	26.6 TFLOPS	0.768 TFLOPS
単精度浮動小数性能	19.5 TFLOPS	51.2 TFLOPS	1.536 TFLOPS
デバイスメモリ	80GB HBM2e	80GB HBM2e	8 channel DDR4
メモリ帯域幅	1935 GB/s	2039 GB/s	204.8 GB/s
最大熱設計電力 (TDP)	300 W	350 W	155 W

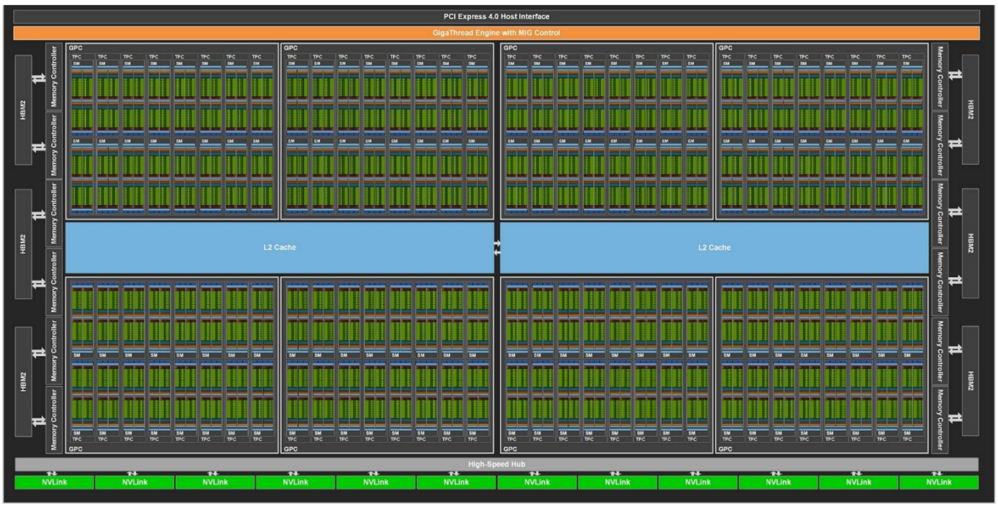

図 2.1: NVIDIA A100 GPU のアーキテクチャ(文献 [10] から引用)

表 2.1 より，GPU である A100 は電力当たりの理論演算性能およびメモリ帯域幅が CPU である EPYC 7302 と比較して非常に高いことが分かる．GPU は，CPU のように Out-of-Order や分岐予測といった命令スケジューリングをハードウェア上で行わない．加えて，すべての命令を SIMD 命令として実装することによって，図 2.1 に示す通り，トランジスタの多くを演算器として利用している．そのため，CPU と比較して高い理論演算性能を有している．しかし，GPU はその高い理論性能を引き出すうえでスループットを最大化するように設計されているため，並列化不可能な逐次処理を行う場合は CPU の方が高速に実行可能である．また GPU は，並列計算を実行することを前提としているため，非常に大きなメモリバス幅を持つことにより，一度に同時ににメモリアクセスを可能にしている．それによって，GPU は CPU と比較してメモリ帯域幅が大きくなっている．

NVIDIA GPU は Streaming Multiprocessor(SM) と呼ばれる演算コアを基本単位として，SM を複数搭載することによって高い並列演算性能を得ている．図 2.2 へ NVIDIA A100 の SM のアーキテクチャを示す．

図 2.2: NVIDIA A100 GPU の SM あたりのアーキテクチャ(文献 [10] から引用)

GPU の SM 内の演算器は，SIMD(Single Instruction Multi Data) プロセッサとして実装されており，非常に多くの実行待ちの SIMD 命令の中から実行可能になった命令を順に実行することによって，DRAM アクセスや命令実行のレイテンシを隠蔽し，SIMD プロセッサとしての実行効率を向上している．またアクティブな SIMD 命令列およびレーンをハードウェアで切り替えられるようになっており，単純な SIMD プロセッサと比較して柔軟にプログラムを実行可能である．こ

のようなハードウェアスレッド実行支援がある SIMD プロセッサを SIMT(Single Instruction Multi Thread) プロセッサと呼んでいる．また NVIDIA GPU において，1 つの SIMD 命令単位を Warp と呼び，その Warp 内の各要素のことをスレッドと呼んでいる．1Warp は 32 スレッドで構成されており，この数のことを Warp サイズと呼ぶ．この Warp は図 2.2 の SM 内で同時に 4 命令分実行が可能である．また SM に割り当てられた各 Warp は共有スクラッチパッドとしても利用可能な L1 キャッシュへアクセスすることができ，SM 内であれば協調して処理を実行することが可能である．

2.3.1 GPU での分岐処理と Warp ダイバージェンス

GPU で各スレッドの分岐命令を処理する場合，CPU のようにプログラムカウンタを変更することによって分岐を実現することはできないため，Warp 内の各スレッドが分岐したかどうかをプリディケートレジスタへ保存しておく．その後，分岐先命令および分岐不成立時の次の命令の両方を実行する．その際に，各スレッドのプリディケートレジスタを参照し，そのスレッドが実行するべき命令ではなかった場合，レジスタ変更やメモリ書き込みなどの状態の変更を行わないことによって，SIMD 命令の各スレッドの分岐命令を仮想的に独立して実行している．このように，SIMD 命令のスレッドの動きがそろっていない実行を Warp ダイバージェンスと呼び，図 2.3 にその実行例を示す．図中において黄緑色の枠が命令を示し，その枠中の白色の破線は演算を行ったがその結果を破棄するスレッド，黒い矢印は実際に実行を行い変更を適用するスレッドである．このように，Warp ダイバージェンスの発生は演算器の実行効率の低下につながるため，その発生を少なくする必要がある．[11]

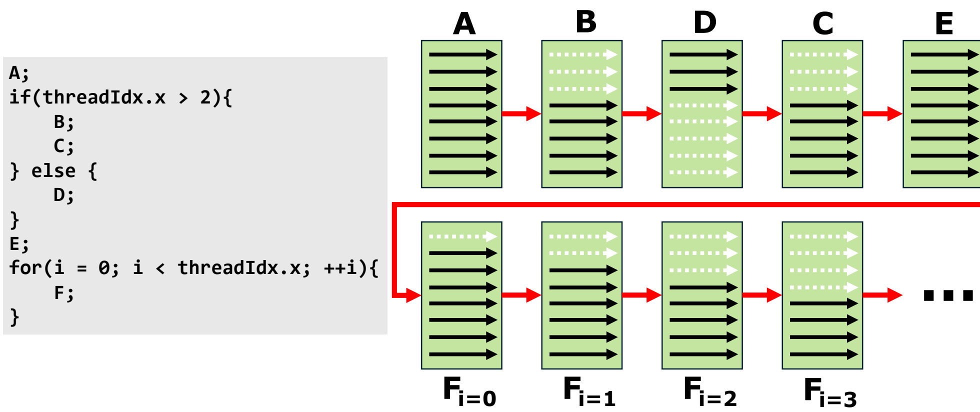

図 2.3: GPU における分岐命令処理により Warp ダイバージェンスが発生している例

2.3.2 Addresss Coalescing とリプレイによるメモリアクセス

また GPU と同じベクトル演算器のベクトルアーキテクチャとの違いとして，ストライドやギャザー・スキャターといったメモリアドレス計算をハードウェアで支援しない点があげられる．できるだけ ROW アドレスの切り替えを行わないようにするため，Addresss Coalescing と呼ばれる，十分な量の転送リクエストがあるまでメモリの転送を遅延し 1 度に転送要求を処理するメモリアクセス手法を採用している．この手法は，メモリ帯域幅の実行効率は改善する代わりにメモリアクセスレイテンシは増加する．そのため，このレイテンシの増加を隠蔽できるだけのスレッド数が必要となる．十分なスレッド数がある場合，この Address Coalescing によってギャザーやスキャターアクセスの DRAM へのアクセスを減らすことができている．[12]

加えて，GPU のロードストアユニットは Cache block と呼ばれる 128Byte ごとにメモリへの転送リクエストを発行している．図 2.4 へリプレイが発生せず，Address Coalescing により 1 回のメモリ転送リクエストで処理できている場合のメモリアクセスを示す．

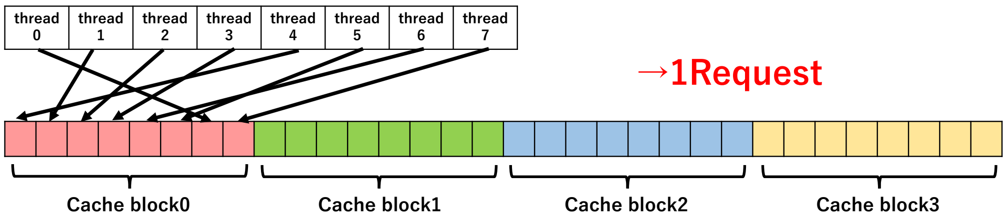

図 2.4: リプレイが発生せず，Address Coalescing により 1 回のメモリ転送リクエストで処理できている場合

Warp 内で非連続な場所へのメモリアクセスが発生した場合でも，そのアクセス範囲が 128Byte の Cache block へ収まっているならば，Address Coalescing によって，メモリ転送リクエストは 1 回で済む．しかし，図 2.5 に示すような Warp 内で連続していないメモリへのロードストア命令を発行した場合，リプレイと呼ばれる複数回のメモリ転送リクエストが発生する．その時のメモリ転送リクエスト回数は最大で Warp サイズである 32 回の 128Byte の転送が必要となり，非常に遅くなってしまう場合がある．[13]

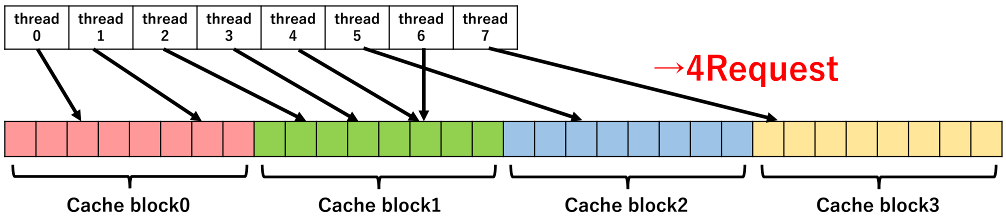

図 2.5: リプレイが発生し，メモリアクセス命令が複数回のメモリ転送リクエストを発生させる場合

2.4 疎行列格納形式

2.2.2 章にて SpMV が反復解法における主要な計算と述べた．その疎行列をメモリ上で表現するにあたり，非ゼロ要素のみを記憶することで大幅にメモリ使用量を減らすことができる．そのような疎行列のメモリへの格納形式を，疎行列格納形式と呼ぶ．SpMV 計算時間は疎行列へのメモリアクセスに大きく依存するため，疎行列格納形式のメモリアクセスが効率的であることが重要視されている．

2.4.1 COO 形式

Coordinate(COO) 形式 [14] は最も単純な疎行列格納形式であり，1 つの非ゼロ要素を値，列番号，行番号の 3 つの組で表現する．図 2.6 へ COO 形式の例を示す．図中の value 配列，col_idx 配列および row_idx 配列は非ゼロ要素の数だけ確保し，非ゼロ要素の値・列番号・行番号をそれぞれ value 配列，col_idx 配列および row_idx 配列へ格納していく．この時の格納順は規定されていない．

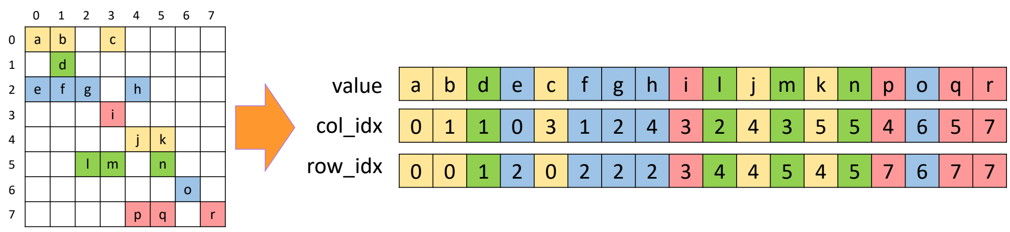

図 2.6: COO 形式への変換例

COO 形式は人が理解しやすく非ゼロ要素の挿入が簡単なため，疎行列の生成や配布時の格納形式として用いられている．しかし，行や列の境界が明確でないため，効率よく SpMV 計算を行うことが困難である．

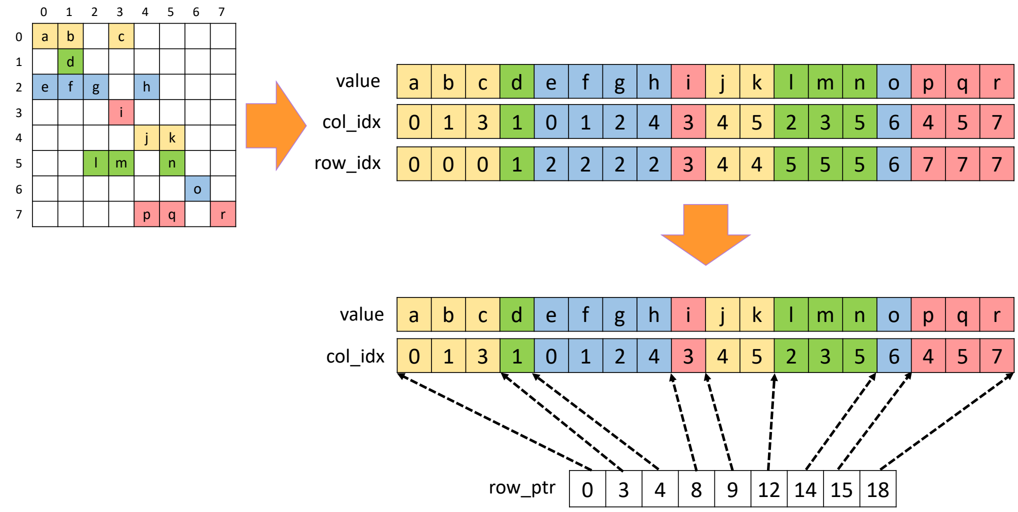

図 2.7: CSR 形式への変換例

2.4.2 CSR 形式

図 2.7 へ Compressed Sparse Row(CSR) 形式 [15] の例を示す．CSR 形式では疎行列の位置情報を行ごとにまとめてメモリへ配置する．COO 形式で保存された疎行列を CSR 形式へ変換する場合，行番号優先で全非ゼロ要素を並び変えた後，COO 形式の row_idx の代わりに各行の非ゼロ要素数の部分和を row_ptr 配列として保存する．この row_ptr 配列は行の開始位置と終了位置として機能する．

CSR 形式は COO 形式と比較して，行番号を圧縮し，行の境界が明確であるという利点が存在する．しかし，非ゼロ要素の追加には挿入する場所より後ろの要素をずらす必要があるため，最大で非ゼロ要素数と同じ回数のメモリ移動が発生する．加えて，列の境界は明確でないため転置疎行列密ベクトル積には不向きである．

この CSR 形式を用いた SpMV の GPU カーネルを Algorithm1 へ示す．また，図 2.8 へ GPU 上での CSR 形式の SpMV 計算時のメモリアクセスを示す．

Algorithm 1 CSR での単純な SpMV

Ensure: $y = Ax$

$tid \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
$total \leftarrow 0$
for $i \leftarrow row\_ptr[tid]$ **to** $row\_ptr[tid + 1]$ **do**
    $total \leftarrow total + value[i] \times x[col\_idx[i]]$
end for
$y[tid] \leftarrow total$

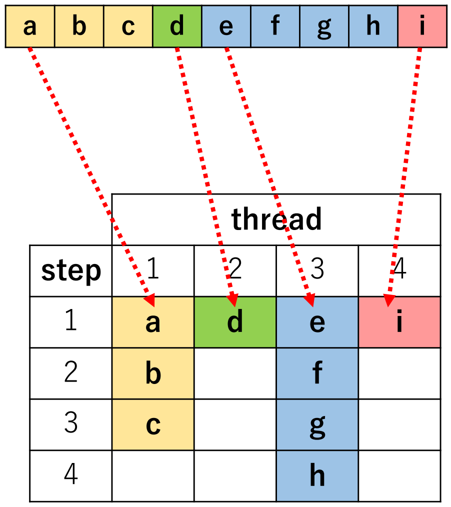

図 2.8: CSR 形式の SpMV 計算時のメモリアクセス

SpMV は疎行列の行ごとにメモリへの書き込みが独立した計算であるため，Algorithm1 では行ごとに並列化することによって GPU の多コアという特性を生かしている．しかし，この計算カーネルの終了は最大の行内非ゼロ要素数に依存する．そのため，行ごとの非ゼロ要素が大きく異なる場合他のスレッドが先に終了し，GPU の過剰なスレッドでメモリアクセスレイテンシを隠ぺいすることができず，大きく実行効率が低下してしまう．加えて，図 2.8 へ示すように，値配列と列番号配列へのアクセスがストライドメモリアクセスになってしまい，このストライド幅が 128Byte を超える場合，リプレイが発生し複数回のメモリ転送リクエストが発行されてしまうため，メモリアクセスペナルティが大きい．

行内非ゼロ要素数の負荷分散については，CSR 形式を用いて GPU 上で高速に SpMV 計算を行う Merge-based SpMV[16] が提案されている．Merge-based SpMV は row_ptr の読み込みと col_idx の読み込みの合計を各スレッドごとに同じになるように，そのマンハッタン距離が等分されるようにスレッド境界を設定することによって，CSR 形式での SpMV の負荷分散を提案している．また，スレッド数が行数に依存しないため，ハードウェアがメモリアクセスペナルティを隠蔽できる十分なスレッド数で実行することができる．NVIDIA の提供している疎行列計算ライブラリである cuSPARSE の CSR 形式を用いた SpMV はこの Merge-based SpMV を用いている [17]．しかし，ストライドメモリアクセスの問題は解決していないため，リプレイの発生によりデバイスのメモリ帯域を完全には引き出せない．

2.4.3 SELL-C-σ 形式

CSR 形式の SpMV のメモリアクセスパターンを改善するために，各要素を列優先で保存した ELL 形式が提案されている [15]．しかし，ELL 形式はメモリアクセス効率を高めるために，不要なメモリパディングが存在するため，CSR 形式と比較してメモリ容量効率が悪い．そこで，行の非ゼロ要素数で行の並び替えを行い隣接行を slice と呼ぶまとまりにすることでメモリパディングを抑えた，メモリ容量効率の良い SELL-C-σ 形式が提案されている [18]．SELL-C-σ の Slice サイズが 4 行の変換例を図 2.9 へ示す．

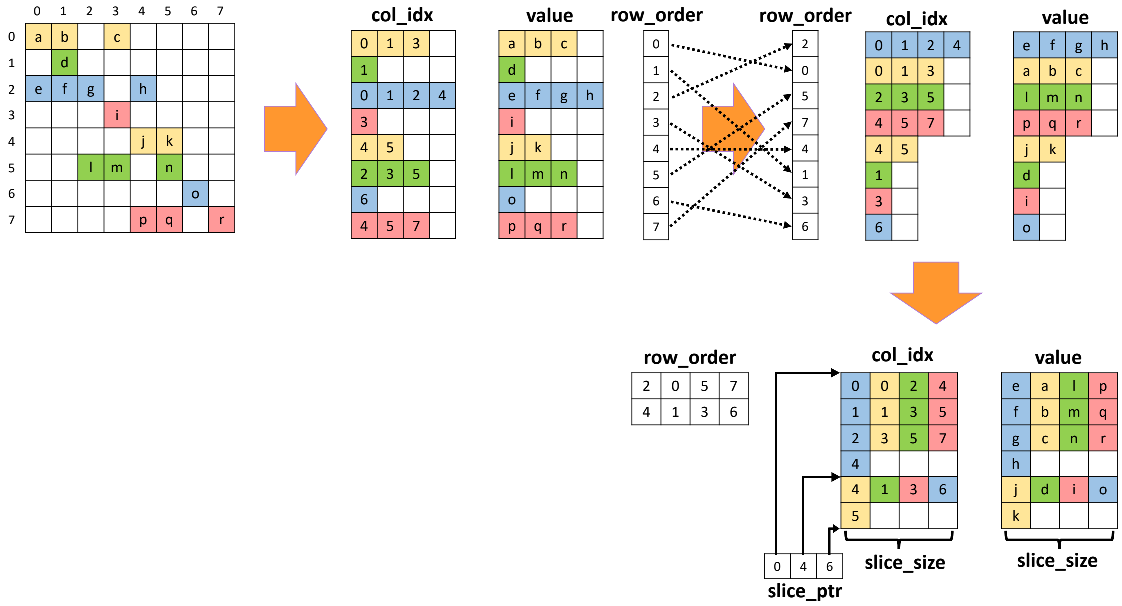

図 2.9: SELL-C-σ 形式への変換例

SELL-C-σ はその要素が複数行ごとに列優先で保存されているため，行の境界が明確である CSR 形式の利点を維持したまま，SIMD 演算器でのメモリアクセス効率を改善している．加えて，Slice にまとめる各行の非ゼロ要素数が近しいため，不要なゼロパディングが少ない．しかし，CSR 形式と同様に列の境界は不明確であり，要素の追加は多くのメモリ移動が発生する問題点が存在している．

SELL-C-σ 形式での SpMV の計算手順を Algorithm 2 へ示す．また，図 2.10 へその計算手順でのメモリアクセスを示す．

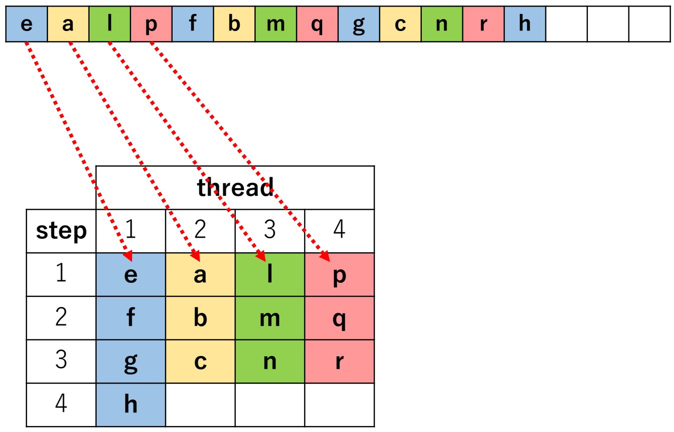

図 2.10: SELL-C-σ の SpMV 計算時のメモリアクセス

SELL-C-σ 形式での SpMV では，各行にスレッドを割り当てることでスレッド間のデータ依存性をなくしている．また，Slice 内の各スレッドがアクセスする要素数が同じとなるため，for ループの終了条件の違いによる Warp ダイバージェンスが起こらない．また，図 2.9 へ示す通り隣接スレッドと連続したアドレスへのメモリ配置となるため，Address Coalescing として Warp 内のメモリからの転送がまとめられ，非連続メモリアクセスによるメモリ転送リクエストのリプレイは密ベクトルへのアクセスのみに抑えられている．

Algorithm 2 SELL-C-σ での SpMV

Ensure: $y = Ax$

$tid \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
$slice\_id \leftarrow tid/C$
$id\_in\_slice \leftarrow tid\%C$
$total \leftarrow 0$
for $slice\_iter \leftarrow slice\_ptr[slice\_id]$ **to** $slice\_ptr[slice\_id + 1]$ **do**
  $i \leftarrow slice\_iter \times C + id\_in\_slice$
  $total \leftarrow total + value[i] \times x[col\_idx[i]]$
end for
$y[row\_order[tid]] \leftarrow total$

Slice あたりの SELL-C-σ のメモリ使用量は以下の式で表される．

$$VCR_{\max} + I(CR_{\max} + C + 1) \tag{2.2}$$

ここで，I は列番号を格納する整数のバイト数，V は値を格納するバイト数，C は Slice サイズ，$R_{\max}$ は Slice 内で最長の行ベクトルの要素数を示す．

C はWarpサイズまたはキャッシュラインサイズに合わせることによりSELL-C-σ ではメモリアクセス効率が良くなることが知られている [19].

2.4.4 CoAdELL

CSR形式とSELL-C-σ 形式でのSpMVは値を倍精度浮動小数として格納し，列番号を32bit整数として格納した場合，メモリアクセス帯域の1/3は非ゼロ要素の位置情報を読み込むことに使用されてしまう．そこでCoAdELLでは，Adaptive ELL(AdELL)と呼ばれるELL形式に対して負荷分散を適用した疎行列格納形式の列番号配列を，その行の最も小さい列番号からの差分として表すことによって列番号のbit数の削減を行っている [20]．これにより，非ゼロ要素の位置情報を圧縮し，疎行列の読み込みに必要なメモリ容量を減らすことでSpMV計算の高速化を達成している．CoAdELLは，疎行列の帯幅が少ない行列に対しては列番号を少ないbit数で表すことが可能であり，差分符号化なため，変換が簡単である利点がある．しかし，可変符号長をGPUで計算しにくいため，最も小さくても8bitの差分符号で格納しなければならない．また，帯幅の大きな行列に対しては差分符号化による圧縮の効果が見られないことが問題である．

2.4.5 PatComp

PatCompは有限要素法を用いたシミュレーションの連立一次方程式の係数行列の非ゼロ要素のパターン性を考慮して，その非ゼロ要素に対して辞書圧縮を適用することによって大幅なメモリ使用量の削減が可能な疎行列格納形式である．SpMV計算時間を高速化するための格納形式ではないが，SpMV計算時間においてもCSR形式を用いた場合とほぼ同じ時間で可能である [21]．しかし，有限要素法に限定している点と辞書を構築するため，その変換時間の大きさが問題である．

2.5 まとめ

有限要素法などの数値シミュレーションは，その要素の多くが0である疎行列を係数行列として持つ連立一次方程式として表される．その連立一次方程式の求解手法として直接解法と反復解法が存在している．直接解法は係数行列の変形を伴い計算量が多いため，大規模な疎行列を係数行列に持つ連立一次方程式の求解には反復解法が用いられている．

反復解法では，SpMV計算がその実行時間において重要な計算であり，メモリ帯域律速であるSpMV計算を高速に計算するには，広いメモリ帯域幅を持つGPUの活用が重要である．GPUはCPUと異なり演算およびメモリ帯域のスループットを最大化する設計がなされており，特徴的な分岐命令実行とメモリ転送リクエ

ストを行っている．

そのような GPU 上で SpMV 計算を効率的に行うためメモリアクセスペナルティや負荷分散に焦点を当てた疎行列格納形式が提案されている．一般的に用いられている疎行列格納形式である CSR 形式を用いた GPU での SpMV 計算はストライドアクセスとなるため，メモリアクセスペナルティが大きい．そのため，GPU 上でメモリアクセスペナルティを小さくした SELL-C-σ 形式が提案されている．しかし，メモリ帯域幅が SpMV 計算の速度上限となることには変わらないため，更なる SpMV 計算の高速化のためには，疎行列へのメモリアクセス回数を減らすことが非常に重要である．

メモリアクセスを減らすため，非ゼロ要素の値そのものを圧縮する手法と，非ゼロ要素位置を圧縮する手法があげられる．両者を比較すると，非ゼロ要素の値を圧縮する場合，対象とする疎行列の値が実数やバイナリ，複素数など行列依存の側面が強い．非ゼロ要素位置の圧縮では整数かつ行番号と列番号の組み合わせがユニークなため，値そのものより圧縮しやすい．そのため，非ゼロ要素の位置情報を圧縮した疎行列格納形式が提案されている．

非ゼロ要素位置を圧縮する手法として差分符号化と辞書圧縮があげられる．差分符号化を適用した CoAdELL 形式では列番号の差分をとり，ビット数を減らすことによって容量を削減している．しかし，その差分の大きさによってはビット数を減らせず，可変長符号は GPU で計算しにくいという欠点が存在している．一方，辞書圧縮方式を適用した PatComp では対象とする疎行列のパターン性によっては非常に高い圧縮率が期待できるが，辞書の単語長が異なると SpMV 計算時に Warp 内のスレッドが実行している命令が異なってしまい，Warp ダイバージェンスと呼ばれる実行効率の低下が発生する問題点が存在している．

第3章　提案手法

3.1　はじめに

本章では，SELL-C-σ形式を元に非ゼロ要素位置情報の辞書圧縮を行ことによって，SpMV計算時の疎行列データを読み込むのに必要なメモリアクセス回数を削減した疎行列格納形式を提案する．行内の非ゼロ要素パターンが近しい複数行の同一な列番号を辞書へまとめることにより，メモリ使用量の削減が期待できる．その時のメモリ配置を工夫することにより，SpMV計算時のメモリアクセス回数が削減され，SpMV計算時間の高速化が期待される．

また，非ゼロ要素パターンが近しい行を探索するため，形式変換時間が長くなることが予測される．そのため，格納形式の変換に必要な計算時間の分析を行い，探索の削減を施した形式変換アルゴリズムの提案を行う．これにより，変換オーバーヘッドが削減され，SpMV計算の高速化の恩恵を受けやすくすることが期待される．

3.2　非ゼロ要素位置辞書圧縮を適用した疎行列格納形式 (Column-indies Dictionary-compressed SELL: CoD-SELL) の提案

図3.1へ提案したCoD-SELL形式の変換例を示す.

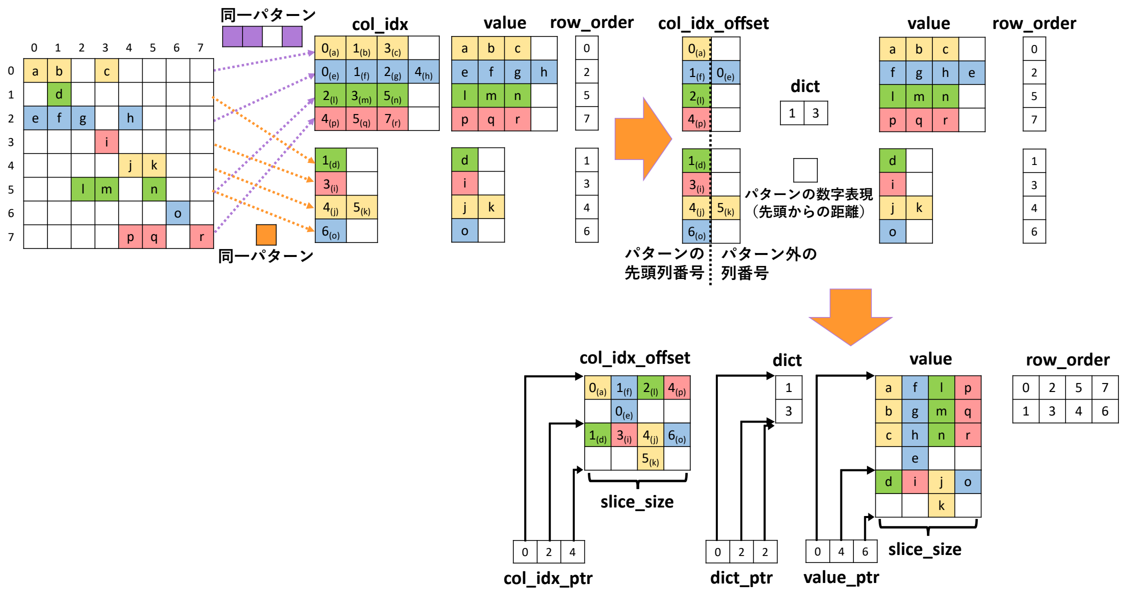

図 3.1: CoD-SELL への変換例

CoD-SELL 形式では，各行について非ゼロ要素パターンが近しい複数行を Slice と呼ばれる区切りでまとめる．図 3.1 では，図中左上紫色の 3 要素の非ゼロパターンを含む行として 0，2，5，7 行目の 4 行を Slice へまとめている．

次に，Slice へまとめた非ゼロ要素の同一パターンの先頭からの距離を dict へまとめ，Slice へまとめた各行の同一パターンが始まる先頭列番号を offset として col_idx_offset 配列の先頭へ格納する．パターン外の列番号については offset を格納した後に col_idx_offset 配列へ格納する．図中の紫色のパターンを例とすると，同一パターンはその先頭および 1，3 進んだ箇所に非ゼロ要素が存在している．そのため，dict には 1,3 が格納される．また，各行の同一パターンの先頭列番号は 0，2，5，7 行目においてそれぞれ 0，1，2，4 列目から非ゼロ要素の同一パターンは始まる．そのため，col_idx_offset 配列の 1 列目にその同一パターンの始まる列番号が格納される．その後，同一パターンではない非ゼロ要素である 2 行目 0 列目の列番号を，col_idx_offset 配列へ格納する．
この際，値配列についても格納を行うが，SpMV 計算を行うとき同一パターンを持つ非ゼロ要素から読み込むため，SELL-C-σ と違い，図中の 2 行目のように値配列の行内の非ゼロ要素の順番が列番号順にならない場合がある．

最後に col_idx_offset 配列および value 配列を SELL-C-σ と同様に Slice サイズの幅を持つ Column-major の配列として保存し，各 Slice の開始位置と終了位置を col_idx_ptr および value_ptr として格納する．加えて，dict 配列は 1 次元配列として保存し，各 Slice が参照するべき開始位置および終了位置を dict_ptr として格納する．

結果として同一パターンを持つ Slice については，同一パターンの列番号が dict 配列へまとめられることにより，位置情報の圧縮を実現している．また，図 3.1 中のオレンジ色の同一パターンが 1 列分しか存在しない Slice は，SELL-C-σ と dict_ptr 以外は全く同じメモリ配置となる．そのため，同一パターンを持つ行が非常に少ない場合でも SELL-C-σ とほぼ同じメモリ使用量で格納可能である．

3.2.1 CoD-SELL での SpMV 計算

SpMV 計算アルゴリズムとメモリアクセス

Algorithm 3 へ CoD-SELL での SpMV の計算方法を示す．また，図 3.2 へ GPU 上での CoD-SELL 形式の SpMV 計算時の列番号を計算するためのメモリアクセスを示す．図 3.2 では紫色の矢印として表されている，Algorithm 3 の 7 行目から 16 行目は同一パターン部の計算を行っている．辞書へのアクセスは各ループにおいて Warp 全体で 1 つの要素にしかアクセスされない．そのため，1 要素を各スレッドへブロードキャストするだけになり，次の辞書の要素へのアクセスも連続したアドレスへ配置されるためキャッシュヒットする．また，単語長も同じとなりループの終端条件の違いによる Warp ダイバージェンスも発生しない．17 行目から 20 行目は同一パターン外の値の計算を行っている．これは図 3.2 では赤色の矢印として表されており，SELL-C-σ と同等の処理を行っている．そのため，隣のスレッドと連続したメモリアドレスへのアクセスとなり，コアレッシングアクセスにまとめられ，メモリの読み込み発行回数を抑えることができる．

以上より CoD-SELL での SpMV 計算での疎行列へのアクセスでは，Stride アクセスのような GPU でメモリアクセスペナルティの大きいアクセスが発生しない．そのため，GPU のメモリ帯域を引き出すことができる．

Algorithm 3 提案格納形式での SpMV

Ensure: $y = Ax$

$tid \leftarrow blockIdx.x \times blockDim.x + threadIdx.x$
$slice\_id \leftarrow tid/C$
$id\_in\_slice \leftarrow tid\%C$
$value\_iter \leftarrow value\_ptr[slice\_id]$
$col\_idx\_iter \leftarrow col\_idx\_ptr[slice\_id]$
$total \leftarrow 0$
if $dict\_ptr[slice\_id + 1] - dict\_ptr[slice\_id] > 0$ **then**
  $col\_idx\_offset = col\_idx[col\_idx\_iter \times C + id\_in\_slice]$
  $total \leftarrow total + value[value\_iter \times C] \times x[col\_idx\_offset \times C]$
  $value\_iter \leftarrow value\_iter + 1$
  for $dict\_idx \leftarrow dict\_ptr[slice\_id]$ **to** $dict\_ptr[slice\_id + 1]$ **do**
    $total \leftarrow total + value[value\_iter \times C + id\_in\_slice] \times x[col\_idx\_offset + dict[dict\_idx]]$
    $value\_iter \leftarrow value\_iter + 1$
  end for
  $col\_idx\_iter \leftarrow col\_idx\_iter + 1$
end if
for $value\_idx \leftarrow value\_iter$ **to** $value\_ptr[slice\_id + 1]$ **do**
  $total \leftarrow total + value[value\_idx \times C + id\_in\_slice] \times x[col\_idx[col\_idx\_iter \times C + id\_in\_slice]]$
  $col\_idx\_iter \leftarrow col\_idx\_iter + 1$
end for
$y[row\_order[tid]] \leftarrow total$

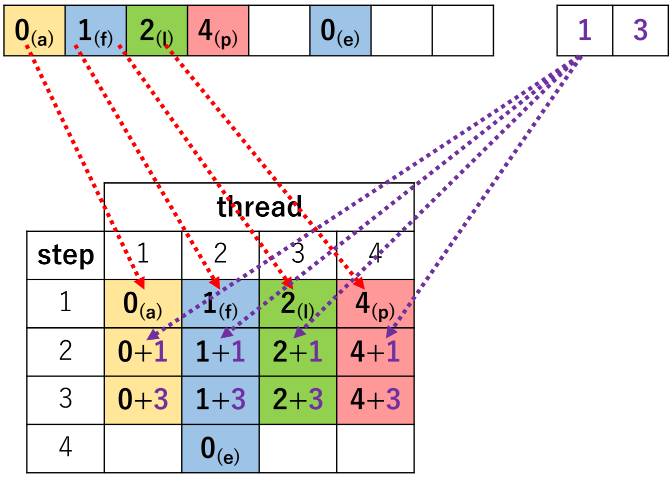

図 3.2: CoD-SELL 形式の SpMV 計算時の列番号のメモリアクセス

理論的な SpMV 計算の高速化倍率

SpMV 計算を行う際，DRAM へ格納された疎行列へのメモリアクセスがその処理時間を占める．そのため，CoD-SELL と SELL-C-σ はメモリアクセスペナルティが非常に少なくメモリ帯域を十分に引き出せるため，SpMV 計算時間は疎行列へのメモリアクセス回数に依存し，メモリアクセス回数はメモリ使用量に大きく依存する．よって，CoD-SELL の理論的な SpMV 計算の SELL-C-σ に対する高速化倍率は

$$\text{SpMV 高速化倍率} = \frac{1}{1 - \text{圧縮率}} \tag{3.1}$$

として表され，SELL-C-σ に対するメモリ圧縮率を求めることにより，CoD-SELL の SpMV 計算の理論的な高速化倍率を求めることができる．

3.2.2　メモリ使用量

もととなる SELL-C-σ の Slice あたりのメモリ使用量は式 2.2 であるが CoD-SELL のメモリ使用量は

$$VCR_{\max} + I((D-1) + C(R_{\max} - D + 1)) + IC + 3I \tag{3.2}$$

で表される．ここで I は列番号を格納する整数のバイト数，V は値を格納するバイト数，C は Slice の行数，$R_{\max}$ は Slice 内で最長の行ベクトルの要素数，D は同一パターンの要素数を示す．

SELL-C-σ からのメモリ削減量はその差分で示され，$I(CD - D - C - 1)$ となる．そのため，CoD-SELL でメモリの使用量が減少する条件は

$$I(CD - D - C - 1) > 0 \tag{3.3}$$
$$CD > D + C + 1 \tag{3.4}$$

となる．また，Slice サイズ C は SELL-C-σ において Warp サイズまたはキャッシュラインサイズに合わせることによりメモリアクセス効率が良くなる．CoD-SELL でも同一パターン外の端数は SELL-C-σ とメモリアクセスパターンが同じなため，C は必ず 2 以上の値をとることになる [19]．そのため，式 (3.4) は $D \geq 2$ で満たすといえる．

また，同一パターンが無い場合 ($D = 0$) でも Slice あたり $I(C+1)$byte のメモリ増加で済むため，最悪の場合でもメモリ増加量が少ない．
圧縮率が最大となる場合は $D = R_{\max}$ であり，Slice あたりのメモリ使用量は式 (3.2) より

$$VCR_{\max} + I(R_{\max} + 2C + 2) \tag{3.5}$$

となる．また，$R_{\max} \gg C + 1$ とするとその時のメモリ容量効率は

$$\frac{VC + I}{VC + IC} = \frac{V}{V + I} + \frac{I}{V + I} \times \frac{1}{C} \tag{3.6}$$

となる．そのため，容量効率は，$C \to \infty$ とした時が理論容量効率となる．しかし，C が大きくなると最大圧縮率も向上するが C 個の行の列番号の同一パターンが少なくなる可能性が急速に高くなる．以上より，C の値は最適な値を探す必要がある．

3.3 形式変換の計算量の削減

CoD-SELL は SELL-C-σ 形式と同様に行ごとに処理することを前提にしているため，行の境界が明確な CSR 形式から CoD-SELL へ変換を行う．その変換手順を以下に示す．

Step 1. 列番号の同一パターンが最長となる行の組み合わせを探索し Slice へまとめる

Step 2. Slice ごとに辞書をメモリへ格納

Step 3. Slice ごとに Column-Major で値配列と列番号の基準と端数をメモリへ格納

Step 1. の辞書を構築するために Slice へまとめる行数を c 行とすると

$$\bigcap_{i\in\{1,2,\cdots,c\}} \{e - b_i \mid e \in v_i, e > b_i\} \tag{3.7}$$

で表される同一パターンの個数が最大となる，c 行の列番号ベクトル $\{v_1, v_2, \cdots v_c\}$ およびその列番号差分の基準となる列番号 $\{b_1 \in v_1, b_2 \in v_2, \cdots, b_c \in v_c\}$ の組み合わせを探索する必要がある．この探索が CoD-SELL の変換において最も計算時間のかかる処理である．

全探索する場合，その理論計算量は n を行列の行数，l を slice にまとめる行の最小非ゼロ要素数 $(\min_{i\in\{1,2,\cdots,c\}} |v_i|)$ とするとその計算量は以下の式として表される．

$$_nC_C \times l^C \propto (nl)^C \tag{3.8}$$

ここで，SpMV 計算高速化のため，C は Warp サイズまたはキャッシュラインサイズというハードウェアに依存した大きさにする必要がある．そのため，SpMV 計算の高速化のためには C は固定値であり，この値を小さくすることができない．そのため，全探索は現実的な計算時間で完了しないことが予測される．

3.3.1 列番号の同一パターンが最長となる Slice の探索の削減

列番号の同一パターンが最長となる行の組み合わせを，COO 形式から CSR 形式への変換と似た計算量で探索するため，Step 1. を以下に示す (i)〜(iii) の 3 段階に分割する．この処理の分割によって，$_nC_C$ の組み合わせを探索することが，$_nC_2$ の組み合わせの探索を $\log_2 C$ 回繰り返すことで近似最適解を求める．また，この処理の分割によって C は 2 の乗数以外の値をとれないが，Warp サイズは 2 の乗数倍であるため問題にはならない．

(i) 各行の非ゼロ要素数で行を並び替え

(ii) 各行について，近傍の行の中から同一パターンが最大となる行のペアを探索

(iii) slice の行数になるまで近傍の Step2 の中から同一パターンが最大となるペア同士をマージしていく

手順 (i) について，各行間の列番号の同一パターンを計算する上で，比較する行の行内非ゼロ要素数が大きく異なると同一パターンの個数は行内非ゼロ要素の少ない方が上限となる．そのため，比較する行の行内最大非ゼロ要素数が大きく異なる場合は非効率となるという仮定の下，並び替え後の近傍数行と比較することで効率的な探索を行えると考えられる．また，並び替えアルゴリズムは行数 n に対して $O(n \log n)$ で計算可能である．

2 行間の最長同一パターンの探索の削減

手順 (ii) では図 3.3 に示すような 2 行間の最長同一パターンを求める必要がある．

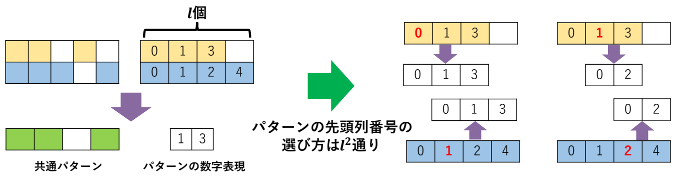

図 3.3: 2 行間の最長同一パターンの計算例

同一パターンを計算するには，基準となる列番号を選択し，その差分の積集合を求める必要がある．基準となる列番号の選択について全探索を行う場合，図 3.3 の各行の要素数を l とすると同一パターンを求める計算量は $O(l^3)$ と計算量が多い．そのため，列番号差分の基準となる列番号の選択を先頭 $\log l$ 個のみに絞ることによって $O(l \log^2 l)$ で 2 行間の列番号の同一パターンを計算できる．これは列番号差分部分の個数が，基準として選んだ列番号より大きい列番号の個数以下になるため，行内の後半部を基準として選ぶことは探索効率が落ちると予測して探索範囲を削減している．その探索削減を適用した 2 行間の列番号の最長同一パターンの計算アルゴリズムを Algorithm 4 に示す．

Algorithm 4 2 行間の列番号の最長同一パターンの計算アルゴリズム

function FIND_LONGEST_PATTERN(a, b)
    $I_{max} \leftarrow \{\}$
    $a_{max\_base}, b_{max\_base} \leftarrow a[0], b[0]$
    for $a_{base} \leftarrow a[0]$ **to** $a[\log |a|]$ **do**
        for $b_{base} \leftarrow b[0]$ **to** $b[\log |b|]$ **do**
            $a_{diff} \leftarrow \{e - a_{base} \mid e \in a, e > a_{base}\}$
            $b_{diff} \leftarrow \{e - b_{base} \mid e \in b, e > b_{base}\}$
            $I = a_{diff} \cap b_{diff}$
            if $|I_{max}| < |I|$ **then**
                $I_{max} \leftarrow I$
                $a_{max\_base} \leftarrow a_{base}$
                $b_{max\_base} \leftarrow b_{base}$
            end if
        end for
    end for
    return $a_{max\_base}, b_{max\_base}, I_{max}$
end function

同一パターンが最長となる行のペアの探索

探索削減を適用した 2 行間の列番号の最長同一パターンの計算アルゴリズムを踏まえて，手順 (ii) である近傍の行の中から同一パターンが最大となる行のペアを探索を行う．手順 (ii) の処理例を図 3.4 へ示す．

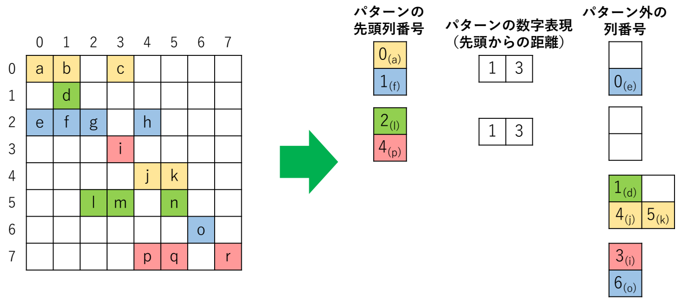

図 3.4: CoD-SELL 形式変換の手順 (ii) の処理例

ここで，手順 (ii) において各行について近傍の数行の中から列番号の同一パターンが最大となる行のペアを重複なく決定していく必要がある．そのため最も簡単な方法として，逐次的に行内非ゼロ要素が多い行からペアの決定を行う．そのアルゴリズムを Algorithm 5 へ示す．ここで Algorithm 5 における R は探索する近傍の行の範囲を決める定数である．

Algorithm 5 手順 (ii) および (iii) にてペアもしくはマージする行の決定アルゴリズム (逐次)

$F_{used} \leftarrow \{false, ...\}$
for $i \leftarrow 0$ **to** n **do**
    $PIdx \leftarrow i$
    $PSize \leftarrow 0$
    for $j \leftarrow i + 1$ **to** $i + R$ **do**
        if $!F_{used}[i]\&!F_{used}[j]$ **then**
            $T = \mathrm{FIND\_LONGEST\_PATTERN}(\mathrm{v_i}, \mathrm{v_j})$
            if $|T.I_{max}| > PSize$ **then**
                $PIdx \leftarrow j$
                $PSize \leftarrow |T.I_{max}|$
            end if
        end if
    end for
    if $PSize! = 0$ **then**
        $\mathrm{MAKE\_PAIR}(\mathrm{v_i}, \mathrm{v_{PIdx}})$
        $F_{used}[i] \leftarrow true$
        $F_{used}[PIdx] \leftarrow true$
    end if
end for

この Algortim 5 の計算量は 2 行間の同一パターンの計算を行数 $\mathrm{n} \times R$ 回行うことになるため，疎行列の非ゼロ要素に依存した計算量としては $O(nl\log^2 l)$ となる．

手順 (iii) においても Algorithm 5 と同様の手順で $\mathrm{FIND\_LONGEST\_PATTERN}(\mathrm{v_i}, \mathrm{v_j})$ の処理を手順 (ii) で既に計算済みの列番号の同一パターンの積集合を求めることによって探索を行う．図 3.5 へその処理の例を示す．

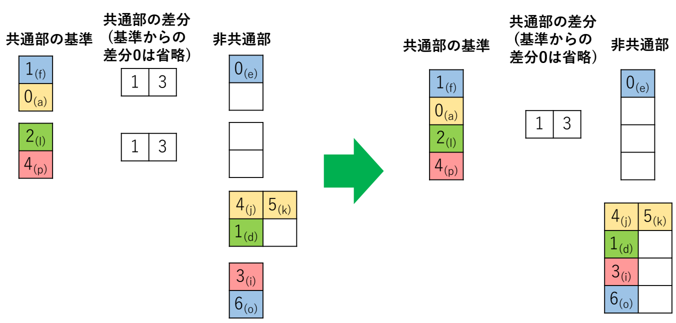

図 3.5: CoD-SELL 形式変換の手順 (iii) の処理例

手順 (iii) では，手順 (ii) で計算しメモ化した列番号の同一パターンの積集合を見ればよいので，マージ対象と列番号の同一パターンの個数を $O(l)$ で計算できる．手順 (iii) 全体では $O(nl)$ で計算可能である．

3.4 まとめ

大規模な疎行列の SpMV 計算の高速化を目的として，SELL-C-σ の列番号配列に対して辞書圧縮を適用した CoD-SELL 形式の提案を行った．同一の非ゼロ要素パターンを持つ行を Slice へまとめ，同一パターンの列番号を辞書として Slice 内で共有し，パターン外の列番号を SELL-C-σ と同様に Column-major で格納する．これによって，SELL-C-σ のメモリアクセスペナルティの少なさを維持まま，メモリアクセス回数が減少することによって，SpMV 計算の高速化が期待される．加えて，メモリ容量が削減されることによってより大規模な疎行列を GPU の少ないデバイスメモリへ格納可能である．

また，提案した CoD-SELL 形式の形式変換方法の計算量の算出を行ったが非現実的な計算量であった．そのため，形式変換に必要な計算量の削減を行い，COO 形式から CSR 形式への変換と似た計算量での変換方法を提案した．形式変換に必要な最大の計算オーダーとしては手順 (ii) の $O(nl\log^2 l)$ の計算量が最も多く，COO 形式から CSR 形式への変換 $O(nl\log nl)$ と似た計算オーダーで変換可能であると予測される．

第4章　提案形式のメモリ使用量およびSpMV計算時間および他形式との比較・評価

4.1　はじめに

本章では，提案したCoD-SELLのメモリ容量効率，SpMV計算時間について生成した疎行列を用いて実際の測定結果と理論値との比較・評価を行う．これにより，3.2.1章に示したSpMV高速化が得られていることを示す．

また，実際のアプリケーションによって生成された疎行列データを用いて提案したCoD-SELL，CSR形式およびSELL-C-σのメモリ容量効率，SpMV計算時間および形式変換時間について比較・評価を行ことにより，他形式との相対的な優位性を示す.

4.2　実験に用いた計算機環境

表4.1および表4.2へ実験で用いた計算機環境を示す.

表4.1: 実験環境(A100)

Host名	SuperA100
CPU	AMD EPYC 7302 × 2
GPU	NVIDIA A100 80GB PCIe
メモリ	DDR4 3200MHz 16GB×16
OS	Ubuntu 22.04 LTS
CUDA	version 12.1
コンパイラ	nvcc, clang

表 4.2: 実験環境 (H100)

Host 名	H100
CPU	AMD EPYC 7313
GPU	NVIDIA H100 PCIe
メモリ	DDR4 3200MHz 16GB×8
OS	Ubuntu 22.04 LTS
CUDA	version 12.1
コンパイラ	nvcc, clang

4.3 CoD-SELL のメモリ使用量および SpMV 計算時間

本章では，提案した CoD-SELL のメモリ容量効率，SpMV 計算時間について生成した疎行列を用いて測定と評価を行う．測定結果に対して，式 3.6 へ示したメモリ容量効率および式 3.1 へ示した理論的な SpMV 計算の高速化倍率との比較を行うことにより，CoD-SELL が 3.2.1 章へ示した SpMV 計算の高速化を達成できることを示す．また，格納に最適な Slice サイズについても検討を行う．

測定には表 4.1 に示した計算機環境を用いた．また，使用する疎行列データは，行内の非ゼロ要素の同一パターンが多い理想的な疎行列として帯行列を使用した．また，同一パターンが少なくほぼ SELL-C-σ として格納されうる疎行列として，帯行列と非ゼロ要素数が同じとなるよう各行内で一様に非ゼロ要素が散らばったランダム疎行列を生成し使用した．行列サイズは NVIDIA A100 の 108 個ある各 SM に対して 1024 スレッド以上割り当て可能な大きさの $2^{17} \times 2^{17}$ とし，各行の非ゼロ要素数は 32 個とした．格納にあたり，値配列は倍精度浮動小数，その他の位置情報を格納する配列は 32bit 整数を使用した．

4.3.1 メモリ使用量および SpMV 計算時間

CoD-SELL のメモリ容量効率，SpMV 計算時間について生成した疎行列を用いて測定と評価を行う．それにより，3.2.1 で示したメモリ容量効率と SpMV 計算時間の高速化倍率の確認を行う．

加えて，CoD-SELL は 3.2.2 章より Slice サイズ C を変化させるとその圧縮率および SpMV 計算時のメモリアクセス効率が変化する．そのため，Slice サイズを 2 〜256 の間で変化させ，SpMV 計算時間およびメモリ容量効率の変化を確認するこ

とにより，最適な Slice サイズを求める．
図 4.1 へその測定結果を示す．

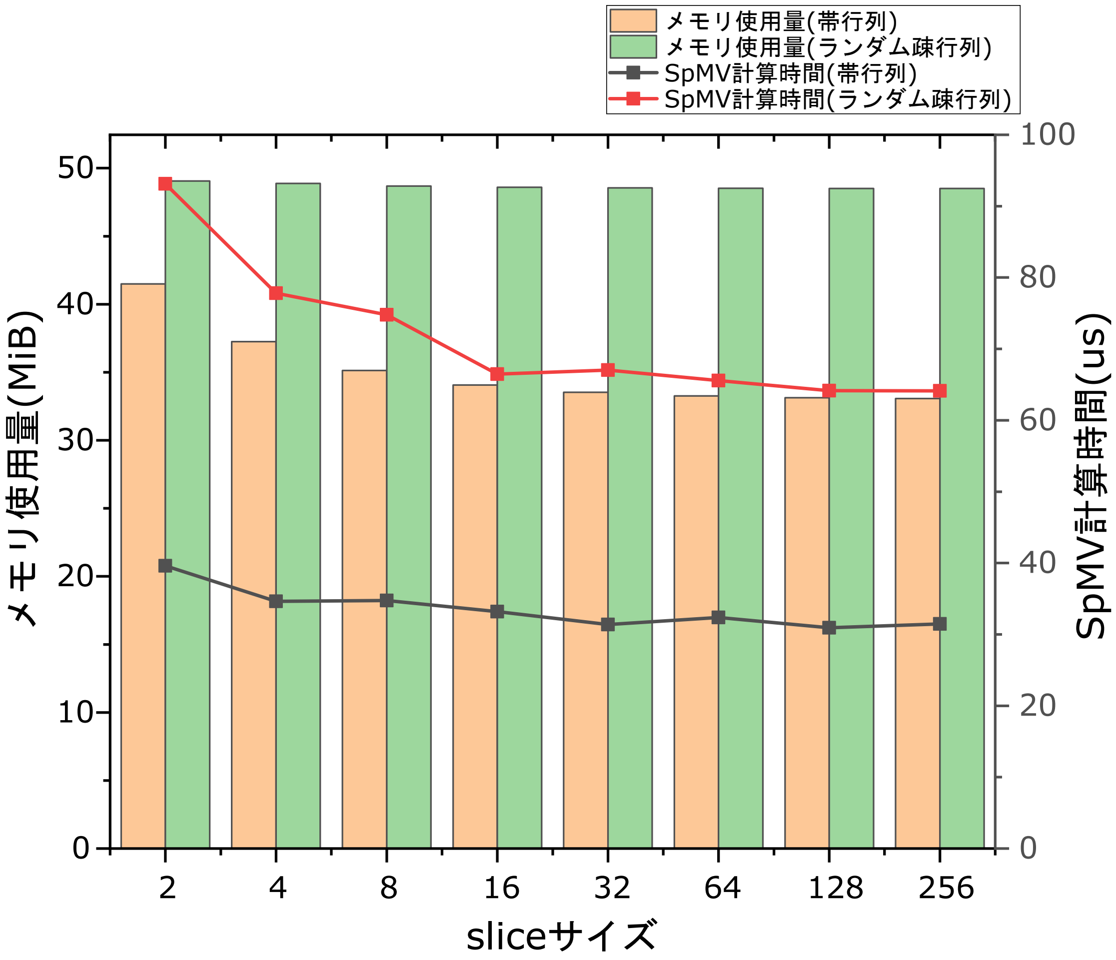

図 4.1: Slice サイズを変化させた際のメモリ使用量と SpMV 計算時間

図 4.1 より，帯行列とランダム疎行列は非ゼロ要素数が同じであるが，CoD-SELL で格納した際のメモリ使用量および SpMV 計算時間に大きな差が見られた．
行内の非ゼロ要素の同一パターンが少ないランダム疎行列では，Slice サイズを変化させたことによるメモリ使用量の大きな削減は見られなかった．しかし，Slice サイズが大きくなるにしたがって SpMV の計算時間が減少している．これは，Slice 単位で連続したメモリアクセスが発生するため，Slice サイズが小さい場合は Warp 内でメモリアクセス範囲が広いため，リプレイが発生していると予測できる．

帯行列では Slice サイズが増加するにつれ，格納に必要なメモリ使用量の減少が見え，ランダム疎行列と比較して最大で 30%以上の削減が得られた．Slice サイズが小さいときはメモリ使用量の減少幅が大きく，Slice サイズが 32 以降は Slice サイズを大きくしてもメモリ使用量の削減効果が薄くなっていることが分かった．また SpMV 計算時間はランダム疎行列と同様に，Slice サイズが 32 までは SpMV 計

算時間が減少し，それ以上の Slice サイズでは変化は見られなかった．

以上より，SpMV 計算時間およびメモリ容量効率の両方で，CoD-SELL の Slice サイズを 32 以上としたするのがよいといえる．

4.3.2 メモリアクセスおよび実行効率のパフォーマンスプロファイリング

本章では，SpMV 計算時のメモリ実行効率やメモリアクセス負荷を確認することにより，前章で述べたメモリアクセスに関する分析を行う．そのため，NVIDIA GPU のパフォーマンスプロファイラである Nsight Compute CLI を用いて計算カーネルのプロファイリングを行った．その結果を表 4.3 へ示す．

表 4.3: 生成した帯行列およびランダム疎行列の SpMV 計算時のメモリプロファイリング結果

測定対象	帯行列		ランダム疎行列	
	slice：32	slice：2	slice：32	slice：2
Elapsed Cycles	30,032	46,435	83,900	121,470
Memory Throughput(TB/s)	1.10	0.964	0.647	0.481
L1 Hit Rate(%)	67.13	67.13	9.93	21.65
L2 Hit Rate(%)	49.72	49.72	62.87	78.08
Branch Efficiency(%)	100	100	100	99.16
total global accesses sectors	2,461,456	5,443,344	5,830,921	8,461,764
uncoalesced global access sectors	151,504	2,723,792	3,139,849	5,744,735

表 4.3 の各疎行列データの Slice サイズ 2 と Slice サイズ 32 の Memory Throughtput を比較すると，どちらも Slice サイズ 32 がメモリ転送効率が高いことがわかる．この原因として，Slice サイズが Warp サイズより小さい場合，total global access sectors が大幅に増加するためメモリ転送効率が低下すると考えられる．これは，Slice サイズが Warp サイズと一致していない場合，複数の Slice へ Warp 内の 1 命令で参照され，連続アクセス幅が小さいリクエストを複数回発行する必要がある．そのため Slice サイズが小さい場合に，total global access sectors が増加していると考えられる．

帯行列およびランダム疎行列の Slice サイズ 32 の結果を比べると，帯行列は L1 Hit Rate の大幅な改善，total global accesses sectors の半減が見られる．L1 Hit Rate の改善は，帯行列は同一パターンへのアクセスが 2 度目以降は L1 キャッシュからロードされているため，ランダム疎行列と比較してキャッシュヒット率が高くなっているためである．total global accesses sectors が半減した理由は帯行列の列番号は同一パターンの先頭以外すべて同一パターンの辞書配列から転送され，そ

の配列は L1 キャッシュからロードされる．また，密行列については倍精度浮動小数が 2^{17} 要素の 1 MiB の配列であるため，A100 の 40MiB の L2 キャッシュに乗り切る．そのため，グローバルメモリへのアクセスはほぼ値配列へのアクセスのみとなり，total global accesses sectors が半減したと考えられる．以上のメモリアクセスの改善によって，SpMV 計算の高速化が達成されていると考えられる．

Branch Efficiency がランダム疎行列の Slice サイズ 2 のみ 100%ではない．これは Slice サイズが Warp サイズより小さく，複数の Slice を 1Warp で処理する際，各 Slice が持つ同一パターン長が異なっているため，Algorithm 3 に示す SpMV カーネルの 16-17 行目にて Warp ダイバージェンスが発生しているためである．

4.3.3 メモリ容量効率および SpMV 計算時間の理論値との比較

3 章で提案した疎行列格納形式である CoD-SELL について，理想的な疎行列として帯行列を生成し，ランダムな列へ非ゼロ要素を挿入した疎行列と比較することでメモリ容量効率および SpMV 計算時間について評価を行った．

測定結果から Slice サイズが 256 の帯行列は，Slice サイズが 2 のランダム疎行列のメモリ使用量の約 67.4%のメモリ容量効率であった．式 3.6 に示した SELL-C-σ との理論容量効率は 66.8%である．理論容量効率と測定結果の容量効率が約 0.6%違う原因として，ランダム疎行列はほぼ SELL-C-σ として格納されると説明したが，同一パターンを一切持たないわけではないため，SELL-C-σ と比較して少ないメモリ使用量であったため，理論容量効率との比較でずれが生じたと考えられる．

SpMV 計算時間は，表 4.3 に示したとおり，2.79 倍の高速化が得られている．しかし，帯行列は密行列へのアクセスが連続アクセスとなるが，ランダム疎行列は密行列へのアクセスがランダムアクセスとなる．そのため，Memory Throughput に差が出ている．ここで，ランダム疎行列でも帯行列と同じ Memory Throughput が出ているとして SpMV 計算時間を算出しなおすと，帯行列ではメモリ容量削減によって 1.64 倍の高速化が達成できているとわかる．式 3.1 へ測定結果のメモリ容量効率の測定値である 67.4%を代入すると，理論的な SpMV 計算時間の高速化倍率は 1.48 倍である．測定結果が理論的な高速化倍率より高い理由として，式 3.3.1 はすべてのメモリ読み込みが DRAM から転送されると仮定しているが，実際は dict 配列が L1 キャッシュから効率的に転送されているため，理論高速化倍率より実際の高速化倍率が高くなったと考えられる．

4.3.4 まとめ

3.2.1 章で説明したメモリアクセス回数削減による SpMV 計算の高速化が達成できていることが確認できた．

加えて，メモリ容量効率および SpMV 計算時間について最適な Slice サイズが 32 以上であることが分かった．

4.4 実際のアプリケーションによる疎行列を用いた他形式との比較・評価

本章では，実際のアプリケーションによって生成された疎行列データを用いて提案した CoD-SELL，CSR 形式および SELL-C-σ のメモリ容量効率，SpMV 計算時間および形式変換時間について比較・評価を行う．それにより，他形式との相対的な CoD-SELL の優位性を示す．

評価にあたり疎行列データとして参考文献 [22] で使用されている疎行列から SuiteSparse Matrix Collection[23][24] で現在入手可能な正方行列を収集した．また追加のデータとして F1 および af_shell9 も収集した．加えて，実際の有限要素法のシミュレーションデータも使用し fu_str，fu_fluid として測定した．その疎行列の行数，非ゼロ要素数，1 行あたりの非ゼロ要素数の平均および行の非ゼロ要素数の最大値を表 4.4 へ示す．

表 4.4: 測定対象の疎行列データ

行列名	行数	非ゼロ要素数	行内非ゼロ要素	
			平均	最大
mc2depi	525,825	2,100,225	3.99	4
af_shell9	504,855	9,046,865	17.92	40
mac_econ_fwd500	206,500	1,273,389	6.17	44
fu_fluid	82,047	1,516,029	18.48	63
cant	62,451	2,034,917	32.58	78
consph	83,334	3,046,907	36.56	81
cop20k_A	121,192	1,362,087	11.24	81
shipsec1	140,874	3,977,139	28.23	102
fu_stru	130,595	6,953,639	53.25	124
rma10	46,835	2,374,001	50.69	145
pwtk	217,918	5,926,171	27.19	180
pdb1HYS	36,417	2,190,591	60.15	204
scircuit	170,998	958,936	5.61	353
F1	343,791	13,590,452	39.53	435

4.4.1　CoD-SELL の容量効率に関する評価

CoD-SELL の他の格納形式との相対的なメモリ容量効率を評価するため，表 4.4 へ示した疎行列データを CSR 形式へ変換を行いそのメモリ使用量を基準として，SELL-C-σ 形式および CoD-SELL 形式についてメモリ使用量の比を算出した．各格納形式の非ゼロ要素の値は倍精度浮動小数とし位置情報には 32bit 整数を使用した．SELL-C-σ および CoD-SELL の Slice サイズは NVIDIA A100 の Warp サイズである 32 とし，行の並び替えは全行を対象とした．CoD-SELL への変換は逐次版である Algorithm 5 で行い，手順 (ii) における探索範囲 R を 4，手順 (iii) におけるマージする探索範囲を 16 とした．図 4.2 へその結果を示す．

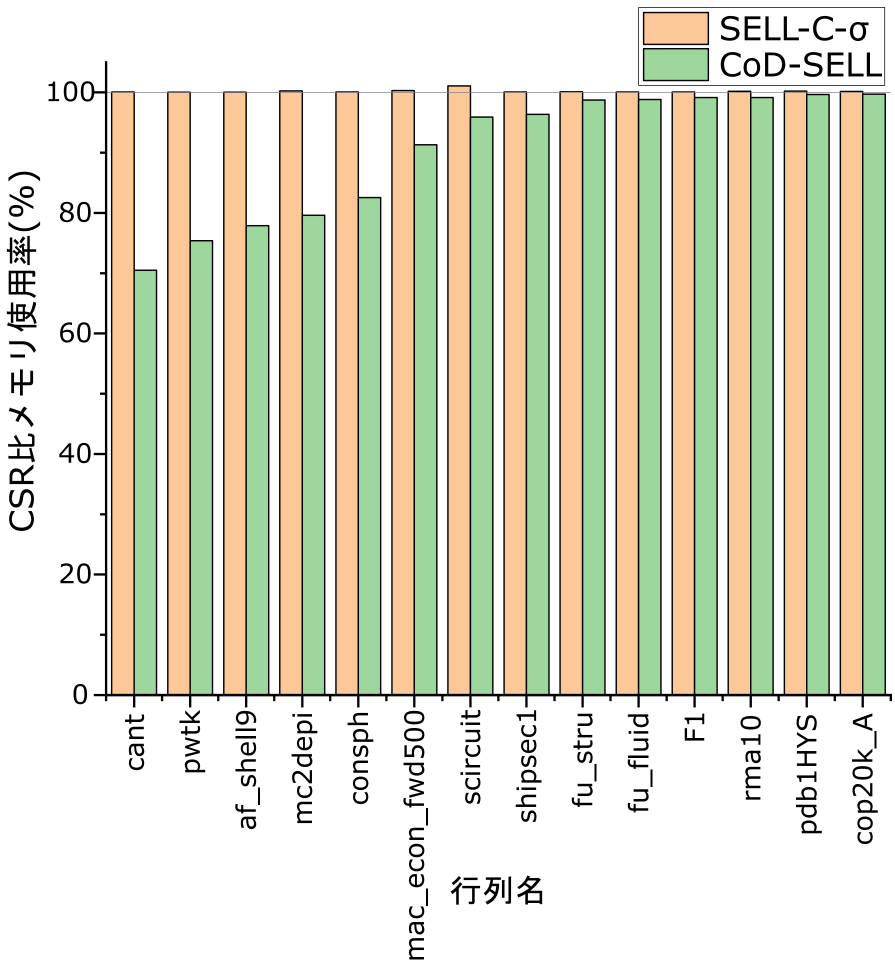

図 4.2: SELL-C-σ と CoD-SELL のメモリ容量効率 (CSR 比)

最もメモリ削減率の高かった cant において CSR 形式比でおよそ 29.5%の容量削減ができた．式 2.2 および式 3.5 より，およそ 30.18%がメモリ削減率の理論値である．そのため，cant は各行の非ゼロ要素のパターンが非常に似通っており，CoD-

SELL において理想的な非ゼロ要素配置であったと考えられる．
SELL-C-σ は Slice 内の各行の要素数が同じになるように 0 埋めを行っている．そのため，すべての疎行列データにおいて CSR 形式よりも 1%未満のメモリ使用量の増加がみられる．対して CoD-SELL は，最も削減率の悪かった cop20k_A についても，CSR 形式よりも少ないメモリ使用量で格納できることが分かった．

4.4.2 SpMV 計算時間に関する評価

各格納形式の GPU 上での SpMV 計算時間を比較・評価を行うため，収集した疎行列を 4.4.1 章と同様に CSR 形式，SELL-C-σ および CoD-SELL への変換を行い，各格納形式での SpMV の計算時間を表 4.1 および表 4.2 に示したマシンで測定した．SELL-C-σ の SpMV 計算には Algorithm 2 を用い，CoD-SELL の SpMV の計算には Algorithm 3 を用いた．計算時間の測定対象は，GPU のカーネル関数の呼び出しから処理が終了しデバイス同期が完了するまでとし，C++標準ライブラリの std::chrono::system_clock で測定した．また，CSR 形式を用いた SpMV の計算には CUDA の疎行列計算ライブラリである cuSPARSE の cusparseSpMV 関数を呼び出してからデバイス同期が終わるまでを std::chrono::system_clock で測定した．また，引数に指定するアルゴリズムは CUSPARSE_SPMV_CSR_ALG1 としたため，内部では Merge-based SpMV が実行されている．

以上の SpMV 計算時間の測定を 30 回繰り返し，その平均値を各形式での SpMV の計算時間とした．以上の実験結果を A100 での結果を図 4.3 へ，H100 での結果を図 4.4 へ示す．

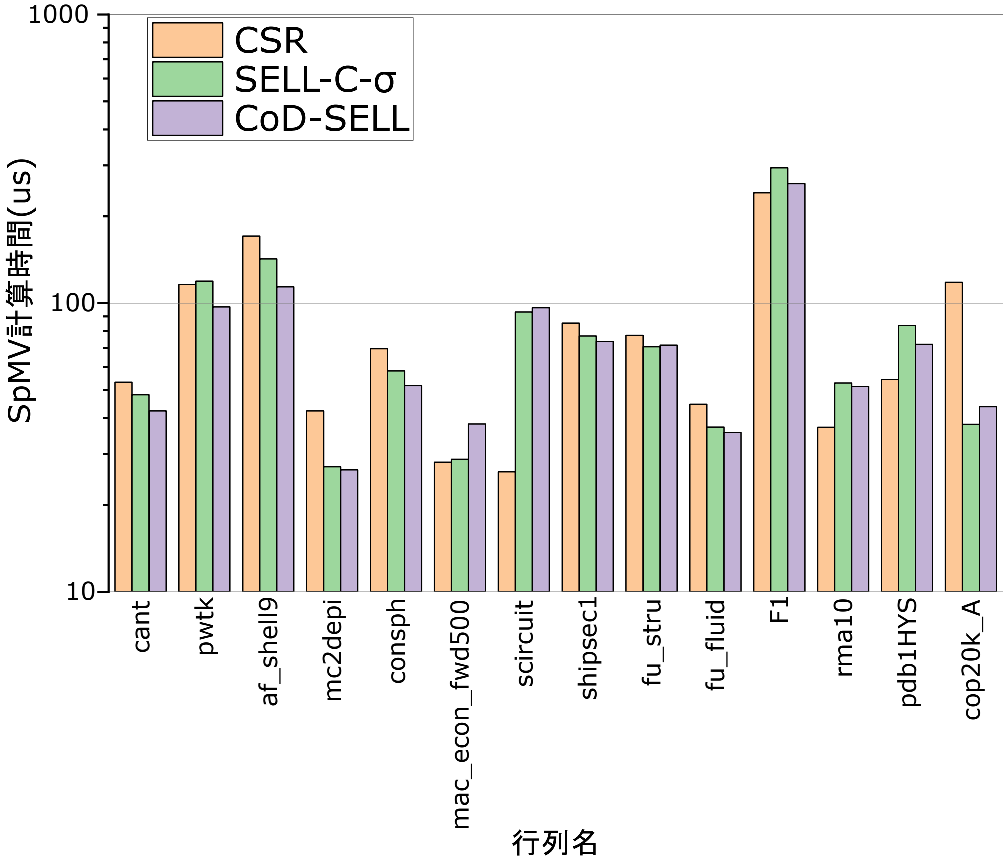

図 4.3: NVIDIA A100 80GB PCIe での各格納形式での SpMV 計算時間 (μs)

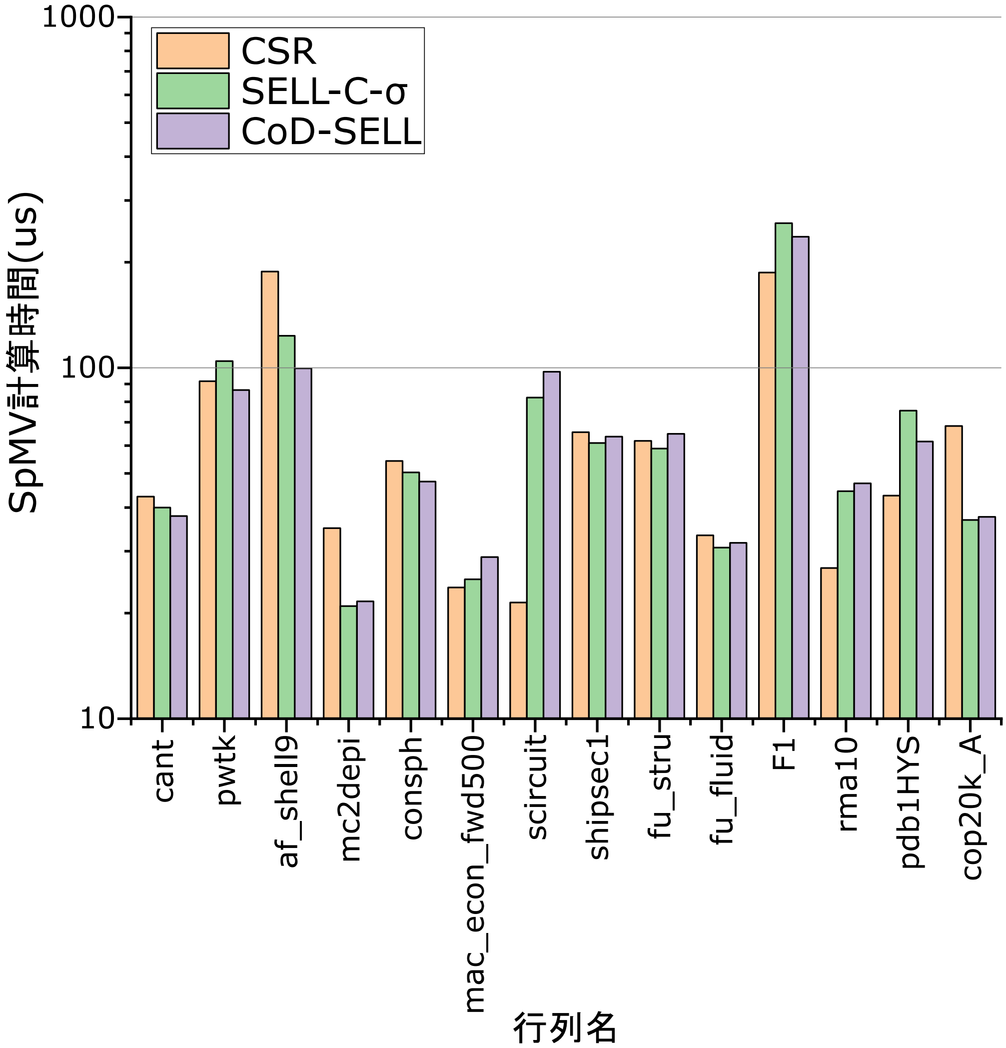

図 4.4: NVIDIA H100 PCIe での各格納形式での SpMV 計算時間 (μs)

圧縮によるメモリ使用量の減少が大きかった cant，pwtk，af_shell9，consph については A100 および H100 の両方で，SELL-C-σ 形式と比較して最大で 19.6%の高速化が得られた．mc2depi は A100 では高速化が見られたが，H100 では SELL-C-σ と比較して遅くなってしまった．A100 80GB PCIe と H100 PCIe のメモリ帯域幅は表 2.1 に示した通りほぼ同じ 2TB/s である．しかし，pwtk および cop20k_A の CSR 形式での SpMV 計算時間が H100 で大幅に短くなっていることから，メモリの間接参照のハードウェアプリフェッチが H100 のほうが精度が良いためであると予測される．

圧縮率の悪い行列に対しては SELL-C-σ と比較して cop20k_A で 23.7%程度遅い結果となった．これはインデックス計算が SELL-C-σ の SpMV 計算では Algorithm 2 の 6 行目のように値配列と列番号配列が同じであるが，CoD-SELL は Algorithm 3 の 18 行目に示した積和演算のインデックス計算が値配列と列番号の場合で異な

り命令数が増えてしまったことが原因であると考えられる．

また，最大行内非ゼロ要素数が平均非ゼロ要素数と大きな差がある疎行列である scircuit，rma10，pdb1HYS，F1 については CSR 形式の SpMV のほうが速い．これは SELL-C-σ とそれを元にした CoD-SELL では，各スレッドが処理を担当する非ゼロ要素数の負荷分散を行っていないためだと考えられる．行内の非ゼロ要素数の平均と最大が極端に違う場合，平均行内非ゼロ要素数までは，ほぼ全スレッドが計算に参加しているためメモリ帯域がボトルネックとなる．しかし，全体の計算の終了は，最大行内非ゼロ要素数で決まり，最大非ゼロ要素数を持つ行を担当するスレッドの計算完了を待つ必要がある．その際，他のスレッドがすでに計算を完了してしまっているため，GPU のメモリアクセスレイテンシの隠蔽がなされないため，その処理の終了が極端に遅くなってしまうためであると考えられる．

4.4.3　CoD-SELL の逐次形式変換の計算時間の評価

3.3.1 章で提案した形式変換の計算量の削減効果を確認するため，CPU のシングルコアにおいて Algorithm 5 の実装を行い，その変換時間を C++標準ライブラリの std::chrono::system_clock で測定した．また CoD-SELL の形式変換時間と比較するため，COO 形式から CSR 形式への形式変換および CSR 形式から SELL-C-σ 形式への変換も CPU 上で逐次的に行いその変換時間を測定した．その結果を表 4.5 へ示す．

COO 形式から CSR 形式への変換では全非ゼロ要素を対象とした並び替えが最も計算量が多く $O(nl \log nl)$ であると理論性能を予測した．表 4.5 の結果より，おおむね非ゼロ要素数 nl の $\log nl$ 倍に比例して変換時間が増加していることが確認できた．しかし fu_stru と rma10，mc2depi については非ゼロ要素数のわりに変換時間が短かった．これは，SuiteSparse MatrixCollection から入手したデータが他の疎行列データでは対称行列として容量を節約した構造で保存されており，COO 形式が上三角と下三角のペアで格納されてしまう．対して fu_stru と rma10，mc2depi は全ての非ゼロ要素がそのまま保存されているため，COO 形式としてメモリ上に展開した時にほぼ並び替えが終わっているためであると考えられる．

CSR 形式から SELL-C-σ 形式への変換では，おおむね非ゼロ要素数 nl に比例して変換時間が増加していることが確認できた．

CSR 形式から CoD-SELL への変換では，非ゼロ要素数の多い F1 や af_shell9 では COO 形式から CSR 形式への変換の 2 倍程度の変換時間であるのに対し，非ゼロ要素数の少ない scircuit や mac_econ_fwd500 では COO から CSR の変換の 3 倍程度の変換時間がかかることが分かった．これは $\log(nl) > \log^2 l$ となるような大きさの疎行列データを使用しているため，計算量が増加したことが理由であると考えられる．

表 4.5: 疎行列データと各格納形式の CPU での変換時間

行列名	非ゼロ要素数	変換時間 (ms)		
		COO →CSR	CSR →SELL-C-σ	CSR →CoD-SELL
scircuit	958,936	82.72	9.92	275.91
mac_econ_fwd500	1,273,389	103.03	15.85	359.78
cop20k_A	1,362,087	274.58	22.39	891.56
fu_fluid	1,516,029	290.79	24.41	724.60
cant	2,034,917	375.24	31.98	713.10
mc2depi	2,100,225	144.31	22.59	636.90
pdb1HYS	2,190,591	416.29	30.08	655.64
rma10	2,374,001	221.72	18.24	401.67
consph	3,046,907	589.12	42.41	987.36
shipsec1	3,977,139	731.30	60.39	1418.67
pwtk	5,926,171	1992.93	76.03	1749.38
fu_stru	6,953,639	805.50	55.32	1678.68
af_shell9	9,046,865	1693.19	151.10	3246.46
F1	13,590,452	3201.08	198.85	6061.69

4.4.4 変換オーバーヘッドを含めた反復回数に関する評価

形式変換に必要な計算時間を SpMV 計算の高速化により償却するのに必要な SpMV の反復回数を評価するため，逐次実装 (CPU) で変換した各格納形式を GPU へ転送し，SpMV の計算時間を測定した．SELL-C-σ および CoD-SELL については CSR 形式の SpMV の計算時間と比較して高速化が確認できた疎行列データに対して，CSR 形式での SpMV 計算時間から各形式での SpMV 計算時間を引くことで計算 1 回あたりの削減できた時間を計算し，その値を CPU 上での逐次実装の変換時間で除算した値を求める．これにより，変換時間をオーバーヘッドとして反復解法に含めた際の疎行列格納形式の変換オーバーヘッドの評価を行う．その結果を表 4.6 へ示す．

SELL-C-σ では平均で 4000 回程度の反復で，CoD-SELL では SELL-C-σ と比較して平均で 9 万回程度と 1 桁ほど多い反復回数で変換オーバーヘッドを SpMV 計算の高速化が上回ることが分かった．CG 法では誤差が含まれない場合は，その係数行列の行数回の反復回数で収束することが知られている．本実験で求めた反復回数をその疎行列データの行数を上回る行列は無かった．しかし，実際にコンピューター上で数値計算を行う場合，浮動小数点が用いられている．浮動小数は表すことが可能な有効数字が有限であるため，計算の際には誤差が含まれている．そのため，理論上の収束に必要な反復回数と実際に必要な反復回数が異なることがある．そのため，実際に反復解法に組み込み，評価する必要があると考えている．

表 4.6: 各格納形式の SpMV 計算時間および変換オーバーヘッドを上回る計算回数

行列名	GPU での SpMV 計算時間 (μs)			$\frac{\text{CSR から削減した時間}}{\text{逐次 (CPU) での変換時間}}$	
	CSR	SELL-C-σ	CoD-SELL	SELL-C-σ	CoD-SELL
scircuit	**26.05**	93.26	96.40	-	-
mac_econ_fwd500	**28.11**	28.77	38.15	-	-
cop20k_A	118.09	**38.06**	43.79	279.8	12000.4
fu_fluid	44.68	37.19	**35.63**	3259.3	80017.6
cant	53.26	48.15	**42.32**	6264.5	65215.8
mc2depi	42.33	27.13	**26.45**	1486.6	40116.7
pdb1HYS	**54.33**	83.62	72.06	-	-
rma10	**37.17**	52.89	51.48	-	-
consph	69.56	58.22	**51.83**	3741.1	55706.3
shipsec1	85.30	76.92	**73.67**	7205.8	122034.3
pwtk	115.98	119.21	**97.00**	-	92194.8
fu_stru	77.37	**70.61**	71.52	8185.6	286924.1
af_shell9	170.88	142.32	**113.93**	5290.1	57007.4
F1	**241.10**	294.24	259.32	-	-

4.4.5 考察

CoD-SELL および他形式のメモリアクセスの考察

2 章および 3 章で説明した CoD-SELL 形式が SELL-C-σ 形式のメモリペナルティが少ないことを維持したまま，メモリ帯域の効率的な利用が行えているかの評価を行うため，表 4.1 に示す計算機環境において，最も SpMV 計算時間の高速化率が高かった af_shell9 について NVIDIA GPU のパフォーマンスプロファイラである Nsight Compute CLI を用いて計算カーネルのプロファイリングを行った．その結果を表 4.7 へ示す．

表 4.7: 各格納形式の SpMV 計算時のプロファイリング結果

測定対象	CSR	SELL-C-σ	CoD-SELL
Elapsed Cycles	189,029	144,027	115,573
Memory Throughput(TB/s)	1.24	1.55	1.49
L1 Hit Rate(%)	40.98	29.95	52.34
L2 Hit Rate(%)	29.11	26.81	29.45
Branch Efficiency(%)	74.13	100	100

表 4.7 の Elapsed Cycles は実行にかかったサイクル数を示しており，CSR 形式，

SELL-C-σ 形式，CoD-SELL の順で実行にかかったサイクル数が少ないことから，CoD-SELL が最も SpMV 計算時間が短かったことと一致している．また，Memory Throughput は実行中のメモリ帯域を示しており，NVIDIA A100 80GB の理論メモリ帯域幅が 1.935TB/s なため，CSR 形式は 63.6%，SELL-C-σ 形式は 81.8%のメモリ帯域効率が出ていることが分かる．CoD-SELL は 79.6%と SELL-C-σ 形式とほぼ同等のメモリ帯域効率を維持している．そのため，3 章で述べたメモリアクセスペナルティの少なさは維持できていると考えられる．加えて，CoD-SELL 形式の辞書へのアクセスは 2 回目以降は L1 キャッシュから行われると説明したが，4.7 の L1 Hit Rate が他の形式と比較して高いことが分かる．

また，Branch Efficiency は 2.3 章で説明した Warp ダイバージェンスが起きていない命令の割合を示している．CSR 形式では，Merge-based SpMV で負荷分散を行っているため，各スレッドが実行する命令でメモリから読み込む要素が列番号配列か row_ptr の 2 パターンがあり，処理している命令が異なる．そのため，Branch Efficiency が低くなっていると考えられる．また，SELL-C-σ 形式および CoD-SELL 形式は Branch Efficiency は 100%となっており，Warp ダイバージェンスが発生していないことが確認できた．

以上より，3 章で説明した CoD-SELL 形式のメモリアクセスペナルティの少なさ，L1 キャッシュから辞書を読み込むことでキャッシュヒット率が向上する点，Warp ダイバージェンスが起きないことが説明できる．

CoD-SELL 形式のメモリ削減率の少なかった行列データに対する考察

提案した CoD-SELL 形式は非ゼロ要素の位置情報の圧縮率に応じた SpMV 計算時間の高速化が可能なことを示したが，af_shell9 の 20%以上メモリ使用量を削減できる疎行列データもあるが，F1 のように 1%以下のメモリ削減しかできていない疎行列データも存在している．この疎行列データによる違いを確認するため，CoD-SELL 形式が有効な疎行列データとして af_shell9，pwtk および cant の非ゼロ要素パターンを可視化した．その結果を図 4.5，4.6 および 4.7 へ示す．

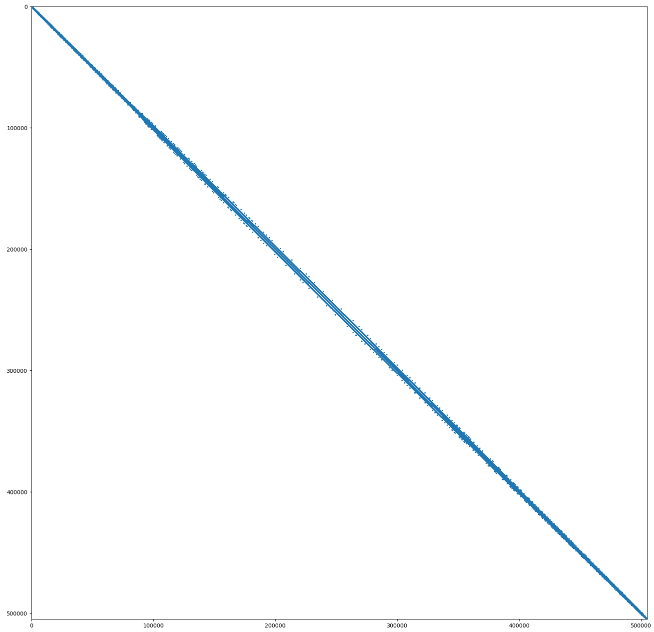

図 4.5: af_shell9 の非ゼロ要素パターン

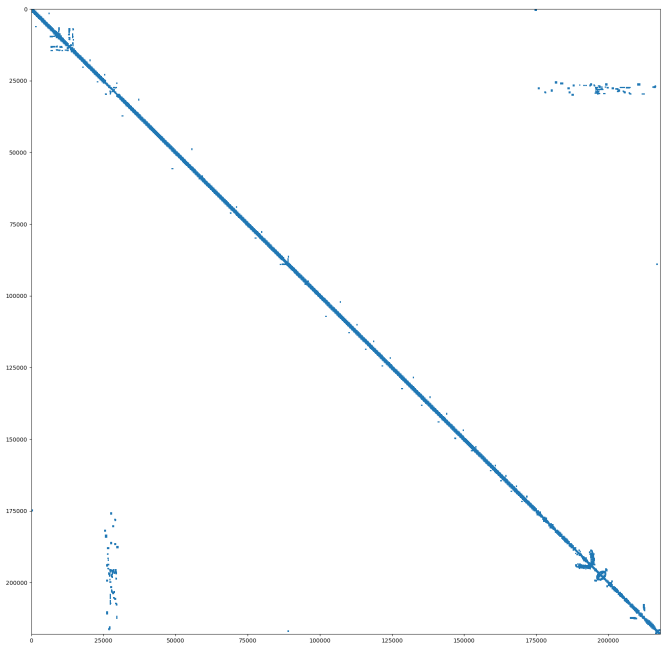

図 4.6: pwtk の非ゼロ要素パターン

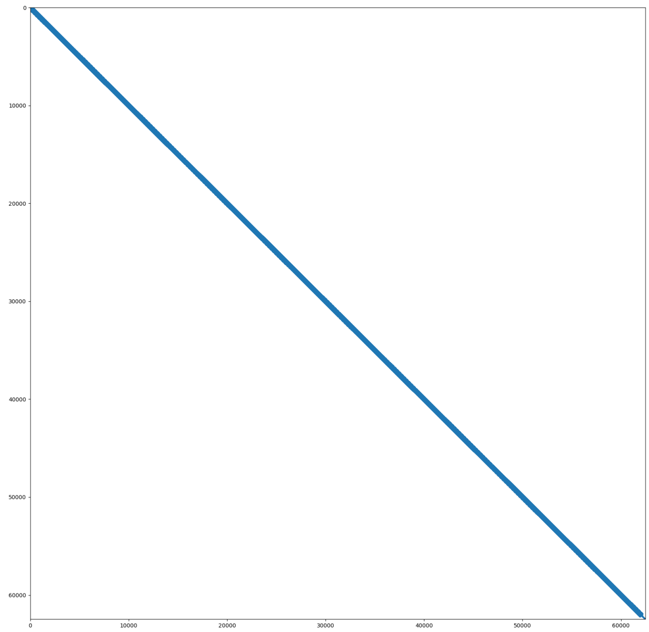

図 4.7: cant の非ゼロ要素パターン

CoD-SELL 形式が有効でない疎行列データとして F1，fu_fluid および fu_str の非ゼロ要素パターンを可視化した．その結果を図 4.5，4.6 および 4.7 へ示す．

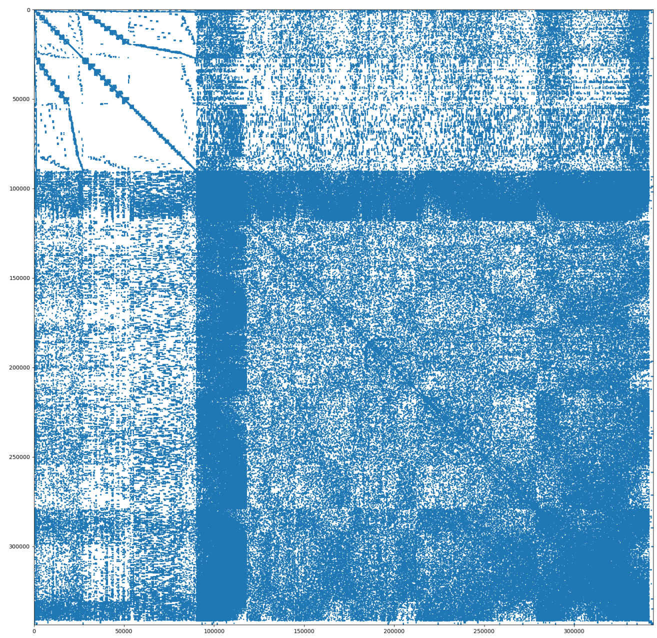

図 4.8: F1 の非ゼロ要素パターン

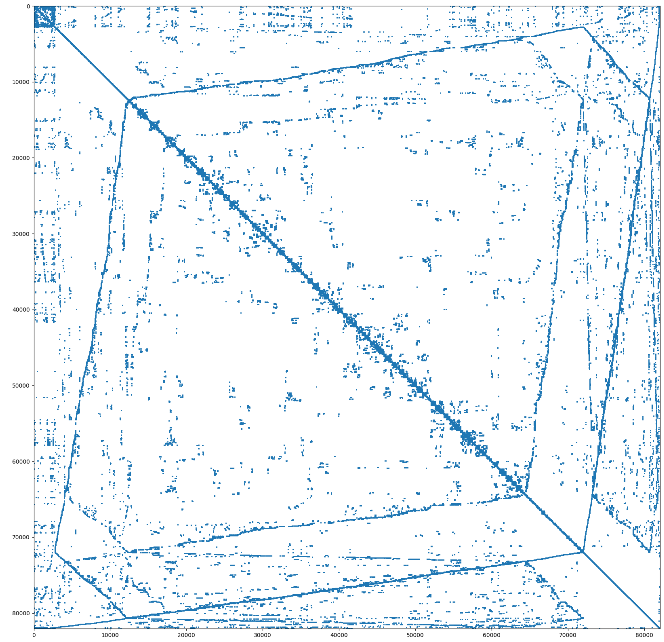

図 4.9: fu_fluid の非ゼロ要素パターン

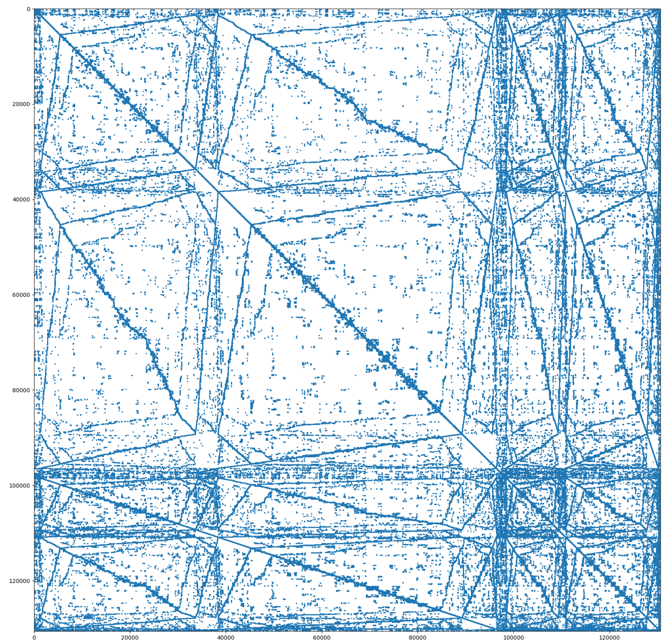

図 4.10: fu_stru の非ゼロ要素パターン

図 4.5，4.6 および 4.7 の非ゼロ要素パターンの可視化結果より，CoD-SELL 形式が有効な疎行列データはその非ゼロ要素の多くが対角の周辺に集中していることが分かった．また，CoD-SELL 形式が有効でない疎行列データはその非ゼロ要素

が全体に分散しており，一部に集中していないため行内の非ゼロ要素のパターンが同じ行が少なくなり，CoD-SELL 形式として格納した際にメモリ圧縮率が悪いと考えられる．

4.4.6 他形式との比較・評価のまとめ

3 章で提案した CoD-SELL について実際のアプリケーションにより生成された疎行列データを用いて，メモリ容量効率，SpMV 計算時間および格納形式変換時間を，CSR 形式および SELL-C-σ と比較し評価を行った．

CoD-SELL は全ての疎行列データにおいて CSR 形式および SELL-C-σ よりも少ないメモリ使用量で格納することが可能であることが分かった．また，行内の非ゼロ要素パターンが似通った行列データにおいて最大で 29.5%のメモリ容量の削減が得られた．

GPU 上での SpMV 計算時間は，メモリ削減率が高い疎行列データにおいて SELL-C-σ のメモリアクセス効率を維持したまま，メモリアクセス回数が減少したことによって最大で 19.6%の高速化が得られた．しかし，メモリ削減率の低い疎行列データにおいては SELL-C-σ より遅くなる場合があることが分かった．

形式変換時間は，逐次的に処理を行う Algorithm 5 では COO 形式から CSR 形式への変換に似た計算量で実現できることが分かった．また，形式変換に必要な計算時間を SpMV 計算の高速化により償却するのに必要な SpMV の反復回数を評価したが，SELL-C-σ と比較して 1 桁多い反復回数が必要なことが分かった．

4.5 まとめ

提案した CoD-SELL において理想的な非ゼロ要素パターンの帯行列を用いて，メモリ使用量および SpMV 計算時間の高速化を確認した．帯行列とランダム疎行列を比較すると，約 67.4%のメモリ使用量で格納することができ，SpMV 計算時間はメモリアクセス回数の削減と L1 キャッシュの効率的な利用により，メモリ実行効率が同じであったと仮定すると 1.64 倍の高速化が得られ，式 3.1 に示した理論値以上に高速化されることが分かった．また，その際の最適な Slice サイズは 32 以上であることが分かった．

加えて，実際のアプリケーションによって生成された疎行列データよるメモリ容量効率，SpMV 計算時間および形式変換時間を CSR 形式および SELL-C-σ と比較することで相対的な評価を行った．結果として SpMV の計算時間ではメモリ使用量の削減に応じた計算時間の削減が見られ，CSR 形式と比較して最大で 19.6%の高速化が得られた．

また，メモリアクセス回数を減らすことを目的として非ゼロ要素の位置情報の圧縮を行ったが，CoD-SELL 形式は全ての疎行列データにおいて CSR 形式および

SELL-C-σ 形式よりも少ないメモリ使用量で格納することが可能であり，CSR 形式と比較して最大で 29.5%のメモリ容量の削減を得られた．また，形式変換時間は，逐次的に処理を行う Algorithm 5 では COO 形式から CSR 形式への変換に似た計算量で実現できることが分かった．加えて，CPU 上での形式変換による変換オーバーヘッドを償却するのに必要な SpMV の反復回数を評価し，SELL-C-σ と比較して 1 桁多い反復回数が必要なことが分かった．

第5章　形式変換の並列化

5.1　はじめに

4.4.4 章において CPU 上での形式変換による変換オーバーヘッドを償却するのに必要な SpMV の反復回数を評価したが，SELL-C-σ と比較して 1 桁多い反復回数が必要なことが分かった．そのため，形式変換を GPU 上で実行することにより形式変換時間の高速化を行い，変換オーバーヘッドを抑えることが期待できる．そこで，本章では提案した CoD-SELL の形式変換を GPU 上で実行するため，並列実行可能な変換アルゴリズムを提案し，その変換時間および探索効率を評価する．

5.2　並列化による GPU での形式変換

3.3.1 章で提案した Algorithm 5 の 6 行目の処理は，直前のループ以前にペアとして選択されているかの判定を行っている．図 5.1 へ逐次的に行のペアの選択をしている例を示す．ペアの選択の際，すでに選択済みの行を避け未選択の行の中から最長同一パターンを持つ行を選択しペアを作成する．新しく行のペアの選択を行うためには，前ループまでの選択済みの行を参照する必要があるためループ伝搬依存性を含んでおり，Algorithm 5 を用いた CoD-SELL の形式変換アルゴリズムは逐次的なアルゴリズムである．GPU は逐次的なアルゴリズムを実行することが不向きであるため，GPU で形式変換を行うには変換アルゴリズムを並列化する必要がある．

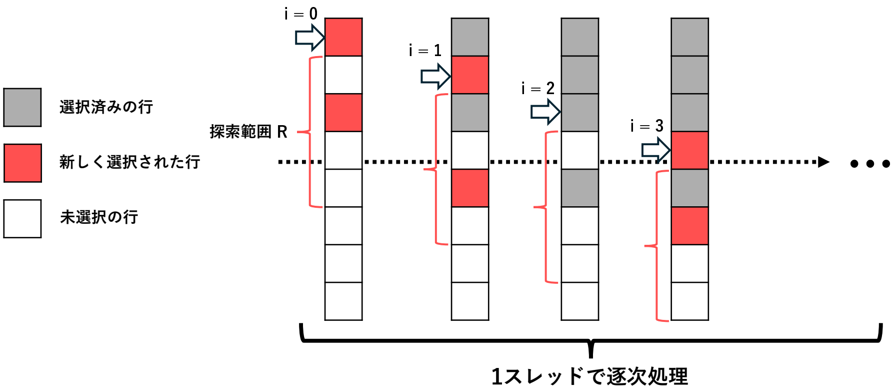

図 5.1: 逐次的に同一パターンが多い行のペアの選択を行っている例

Algorithm 5 において，2 行目から 19 行目のループは，すでにペアとして選択された行を重複して選択しないようにフラグを管理しているため，前のループとの依存性が発生している．そのため並列化の方針として，各行が独立して列番号の同一パターンが最長となる行を，重複を考慮せずに列挙することによりループ間の依存性がなくなる．そのままでは，ペアの選択に重複が発生した状態なため，各行の同一パターンが最長となる行番号を CPU へ転送し，逐次的に重複を排除する．そのアルゴリズムを Algorithm 6 へ，図 5.2 へ並列で最長同一パターンとなる行を列挙し逐次的に行のペアの選択をしている例を示す．

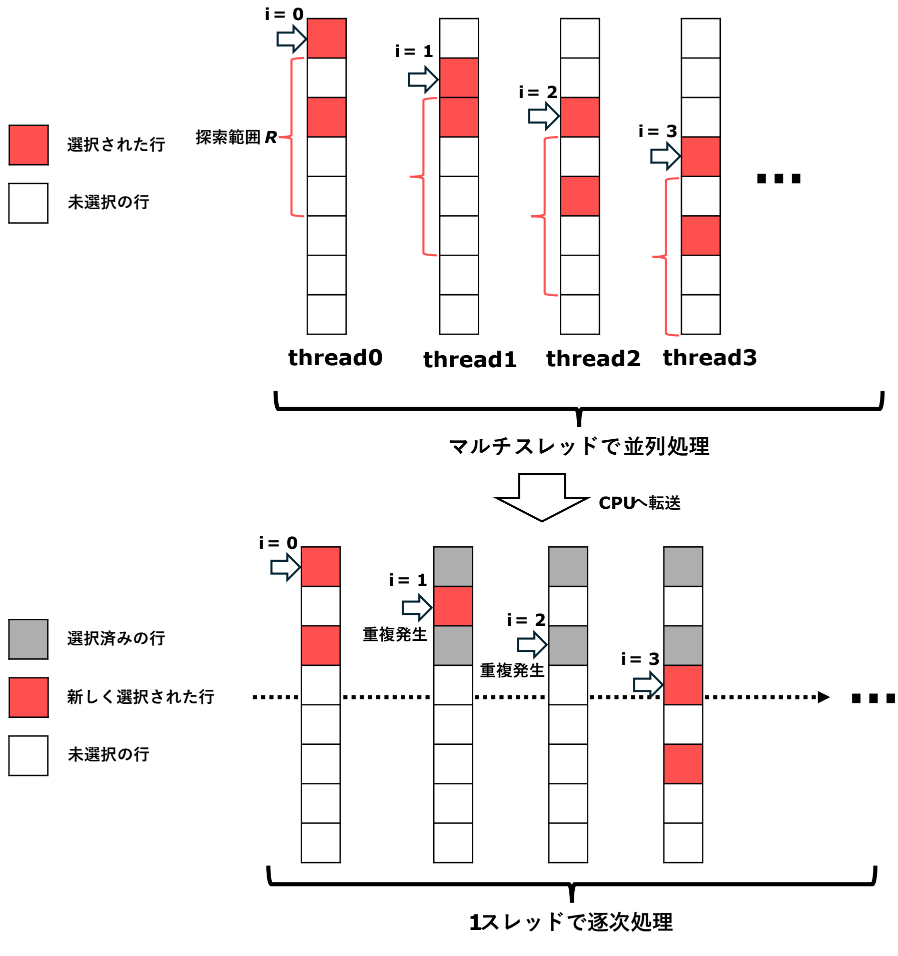

図 5.2: 並列で同一パターンが最長となる行を独立して列挙し，逐次的に行のペアの選択をしている例

Algorithm 6 中の 2 行目から 10 行目のループにおいて，図 5.2 の上側へ示したように各行が独立して並列に同一パターンが最長となる行の探索を行う．そこで得られた行のペアを Algorithm 6 中の 12 行目から 19 行目のループにおいて，図 5.2 の下側へ示すように逐次的に行の重複を排除する．このアルゴリズムの中で FIND_LONGEST_PATTERN を (行数 $\times R$) 回呼び出すことが最も計算量の多い処理である．この処理を並列化することにより，重複を排除する処理は逐次的に行

う場合でも十分に並列化の恩恵を受けられると期待される．

Algorithm 6 手順 (ii) および (iii) にてペアもしくはマージする行の決定アルゴリズム (並列)

$P_{longest} \leftarrow \{\}$
for $i \leftarrow 0$ **to** n **do parallel**
  $PSize \leftarrow 0$
  for $j \leftarrow i+1$ **to** $i+R$ **do**
    $T = \mathrm{FIND\_LONGEST\_PATTERN}(\mathrm{v_i}, \mathrm{v_j})$
    if $|T.I_{max}| > PSize$ **then**
      $P_{longest}[i] \leftarrow j$
    end if
  end for
end for
$F_{used} \leftarrow \{false, ...\}$
for $i \leftarrow 0$ **to** n **do sequential**
  $j \leftarrow P_{longest}[i]$
  if $!F_{used}[i] \& !F_{used}[j]$ **then**
    $\mathrm{MAKE\_PAIR}(\mathrm{v_i}, \mathrm{v_j})$
    $F_{used}[i] \leftarrow true$
    $F_{used}[j] \leftarrow true$
  end if
end for

並列で処理を行う Algorithm 6 の問題点として，最長同一パターンを持つ行として選択したい行が重複している場合，その片一方の行は重複しているとしてペアを作ることができない．逐次変換である Algorithm 5 では図 5.1 へ示した通り，順に選択を行うためペアにしたい行の重複は起きない．そのため，ペアとして選ぶ行の重複があった場合，図 5.2 へ示すようにペアとして選択されず，辞書を構築することができない．逐次変換では，前のループで既に選択されていた行を事前に跳ばすことができるため，2 番目に長い同一パターンを持つ行を選択することができる．そのため，Algorithm 6 の探索効率は逐次変換である Algorithm 5 と比較して悪く，最終的なメモリ容量削減率は悪化すると予測される．

5.3 CoD-SELL の GPU での形式変換時間の評価

5.2 章で述べた形式変換の並列化および GPU での実行時間を評価するため，表 4.1 に示したマシン上で Algorithm 6 を用いて CoD-SELL の形式変換時間を測定し

た．加えて，COO 形式から CSR 形式への形式変換および CSR 形式から SELL-C-σ 形式への変換も GPU 上で行いその変換時間を CoD-SELL の形式変換時間と比較した．その実験結果を 5.1 へ示す．

表 5.1: 各格納形式の並列実装における GPU での変換時間

行列名	非ゼロ要素数	変換時間 (ms)		
		COO →CSR	CSR →SELL-C-σ	CSR →CoD-SELL
scircuit	958,936	1.887	1.164	33.827
mac_econ_fwd500	1,273,389	2.218	1.107	11.748
cop20k_A	1,362,087	4.369	1.009	11.661
fu_fluid	1,516,029	3.779	1.130	10.038
cant	2,034,917	4.387	0.980	18.476
mc2depi	2,100,225	3.144	1.315	20.466
pdb1HYS	2,190,591	4.636	1.093	25.717
rma10	2,374,001	3.462	0.979	14.039
consph	3,046,907	5.488	1.186	38.316
shipsec1	3,977,139	6.431	1.567	44.232
pwtk	5,926,171	8.775	1.768	83.175
fu_stru	6,953,639	6.146	1.350	24.373
af_shell9	9,046,865	12.476	2.112	79.758
F1	13,590,452	18.521	3.150	129.471

表 5.1 の COO 形式から CSR 形式への変換時間は表 4.5 と比較して平均して約 106 倍の高速化が得られた．これは，COO 形式から CSR 形式への変換に全非ゼロ要素の並び替えという計算律速な計算が大きく高速化されたためである．また，CSR 形式から SELL-C-σ への変換時間は GPU 上で変換することによって約 34 倍の高速化が得られたが，COO 形式から CSR 形式への変換の高速化倍率よりも低かった要因として SELL-C-σ への変換は row-major から column-major へのメモリコピーが主な処理なため，メモリ帯域幅の差が大きく出たためであると考えられる．

CSR 形式から CoD-SELL 形式での変換では平均して 39 倍ほどの高速化であった．CoD-SELL への変換は COO 形式から CSR 形式と同様に計算律速であるが，最長共通パターンとなる行の探索の際に重複を排除する処理を，CPU へ転送して逐次的に行っており，GPU 内で完結していない．そのため，転送オーバーヘッドや部分的な逐次処理が原因で高速化率が大きくないと考えられる．

5.4 提案形式の変換実装の違いによる容量効率

5.2 章にて，並列実装は辞書圧縮から漏れてしまう行があり圧縮効率が悪化すると予測した．その逐次実装 (Algorithm 5) および並列実装 (Algorithm 6) での圧縮効率を測定するため，Slice サイズを 2 および 4 とした場合の提案形式のメモリ使用量の CSR 形式比を測定した．

Slice サイズを 2 にした場合，ペアにした段階で Slice へまとめる行数を満たしているため，手順 (iii) のマージを行わない．そのため，手順 (ii) における探索の効率を間接的に測定することができる．Slice サイズを 4 にした場合についても同様に，1 回のみ行のペア同士をマージした段階で Slice へまとめる行数を満たす．その時の CSR 形式とのメモリ使用率を比較することで，手順 (iii) の実装の違いによる探索効率を間接的に測定する．以上の測定の結果を表 5.2 へ示す．

表 5.2: 提案形式の変換実装の違いによるメモリ使用率 (CSR 比)

行列名	CSR 比メモリ使用率 (%)			
	手順 (ii) まで (Slice:2)		手順 (ii) & 手順 (iii)1 回 (Slice:4)	
	逐次実装	並列実装	逐次実装	並列実装
scircuit	103.2	103.4	99.4	102.9
mac_econ_fwd500	99.2	101.5	94.0	102.6
cop20k_A	96.0	91.5	98.6	101.0
fu_fluid	96.4	95.1	98.3	100.5
cant	84.7	84.7	76.4	94.3
mc2depi	100.2	100.1	88.9	93.5
pdb1HYS	90.5	89.6	94.7	100.0
rma10	91.1	88.5	93.7	100.0
consph	85.4	84.9	79.1	88.6
shipsec1	88.2	84.6	88.0	97.9
pwtk	85.1	84.8	77.5	86.5
fu_stru	93.4	92.1	97.9	100.2
af_shell9	86.2	88.6	78.9	90.8
F1	92.0	90.0	97.6	100.2

表 5.2「手順 (ii) のみ (Slice=2)」の容量効率の測定結果から，逐次実装と並列実装に容量効率の違いがほぼないため，ペアとする行の探索においては並列実装でよいことが分かる．手順 (ii) および手順 (iii) を 1 回行う場合 (Slice=4) の測定結果から，並列実装ではすべての疎行列データにおいて逐次実装と比べ，非常に容量効率が悪いことがわかった．これは，最大となるペアのみを記録し，重複によっ

てマージできずに辞書を構築できない行のペアが非常に多いためであることが考えられる．そのため，提案した GPU での CoD-SELL 形式への変換手法を，GPU で高速に実行可能なまま CPU の逐次実装と同じ容量効率となるように改良する必要があると考えている．

5.5　まとめ

4.4.4 章において評価を行った形式変換による変換オーバーヘッドを償却するのに必要な SpMV の反復回数を削減するため，CoD-SELL の形式変換を並列化し GPU 上で実行可能な形式変換アルゴリズムを提案し，変換時間および探索効率を評価した．

GPU 上で形式変換を行うことにより，平均して約 39 倍の高速化が見られた．しかし，並列化するうえで行のペアの選択の重複を考慮していないため，すべての疎行列データにおいてメモリ容量効率が悪化することがわかった．これは行のペアの選択における重複により，辞書を構築できない行が多いことが理由であると考えられる．

第6章　おわりに

6.1　本研究の概要と成果

本研究では，物理シミュレーションを連立一次方程式として記述した際の係数行列である疎行列において，疎行列密ベクトル積 (SpMV) を GPU 上で高速に計算を行うことを目的とし，SELL-C-σ 形式の非ゼロ要素の位置情報を圧縮した CoD-SELL 形式を提案した．また，CoD-SELL 形式の高速な形式変換アルゴリズムも提案し，CoD-SELL 形式の利用に必要なオーバーヘッドの評価を行った．

2 章では物理シミュレーションにおける連立一次方程式および反復解法による求解，その係数行列である疎行列の格納形式を説明した．SpMV を高速に計算するため，広いメモリ帯域を持つ GPU 利用されており，GPU 上で効率的に SpMV を実行するために提案された，CSR 形式を用いた Merge-base SpMV および SELL-C-σ 形式を説明した．CSR 形式および SELL-C-σ 形式では倍精度浮動小数を値として格納し，その位置情報を 32bit 整数として格納した場合，そのメモリアクセス帯域の 1/3 を位置情報の読み込みが占める．そのため，この位置情報を圧縮し SpMV 計算時のメモリアクセス回数を減らすことが重要である．位置情報を圧縮した疎行列格納形式として CoAdELL 形式が提案されているが，CoAdELL 形式は差分符号化を行っているため，行列の帯幅に大きく依存してしまう問題点がある．また，位置情報の辞書圧縮を適用した疎行列格納形式として PatComp 形式が提案されているが，多くの疎行列データで ELL 形式よりも SpMV 計算時間が増加してしまっている．

以上の点を踏まえて，3 章では SELL-C-σ 形式の列番号配列に対して辞書圧縮を適用することで，SpMV 計算時のメモリアクセス回数を減らした CoD-SELL 形式を提案した．CoD-SELL では，非ゼロ要素の位置情報のパターンが似ている行を Slice へまとめ，Slice 内の行の非ゼロ要素の位置情報の共通パターンを 1 つの辞書としてまとめている．また，この辞書は他の Slice と共有せず，非共通パターンは SELL-C-σ と同様に Column-major として格納することにより，SELL-C-σ の SpMV 計算時のメモリアクセスペナルティの少なさを維持したまま，非ゼロ要素の位置情報のメモリアクセス回数の削減を可能としていることを述べた．

4 章では，提案した CoD-SELL において理想的な非ゼロ要素パターンの帯行列を

用いて，メモリ使用量および SpMV 計算時間の高速化を確認した．帯行列とランダム疎行列を比較すると，約 67.4%のメモリ使用量で格納することができ，SpMV 計算時間はメモリアクセス回数の削減により 1.64 倍の高速化が得られ，L1 キャッシュの効率的な利用により式 3.1 に示した理論値以上に高速化されることが分かった．また，その際の最適な Slice サイズは 32 以上であることが分かった．

加えて，提案した CoD-SELL 形式によるメモリ削減率，SpMV 計算時間および形式変換時間を CSR 形式および SELL-C-σ と比較することで相対的な評価を行った．結果として SpMV の計算時間ではメモリ使用量の削減に応じた計算時間の削減が見られ，CSR 形式と比較して最大で 19.6%の高速化が得られた．また，メモリアクセス回数を減らすことを目的として非ゼロ要素の位置情報の圧縮を行ったが，CoD-SELL は全ての疎行列データにおいて CSR 形式および SELL-C-σ よりも少ないメモリ使用量で格納することが可能であり，CSR 形式と比較して最大で 29.5%のメモリ容量の削減を得られた．また，形式変換時間は，逐次的に処理を行う Algorithm 5 では COO 形式から CSR 形式への変換に似た計算量で実現できることが分かった．

5 章では，形式変換の並列化による GPU 上での形式変換時間の評価を行った．CoD-SELL の形式変換を GPU 上で実行することによって約 39 倍の高速化が得られることが分かった．しかし，並列化するうえで重複を排除する処理を簡易的に行ったため，行のペアの選択における重複により，辞書を構築できない行が多く，メモリ削減率が大きく低下してしまうことが分かった．

6.2 今後の課題

CoD-SELL 形式への変換において，並列実装ではすべての疎行列データにおいて，選択する行の重複による辞書を構築できない行が多く，容量効率の低下がみられた．そのため，GPU 上で並列実行可能かつ圧縮率の低下しない形式変換手法を提案する必要がある．

また，提案した CoD-SELL においてメモリ削減率が悪い行列があるが，4.2 および 5.2 の結果から Slice サイズが小さい方がメモリ削減率が高い行列データが存在することが分かった．しかし，Slice サイズは GPU の warp サイズに合わせなければ実行効率が低下してしまう．そのため，warp サイズよりも小さい Slice サイズで実行効率の低下しない SpMV 計算手法について今後研究に取り組みたいと考えている．

加えて，実際に反復解法に CoD-SELL 形式を用いた SpMV を組み込むことで，変換オーバーヘッドを含めた連立一次方程式の求解全体で，計算時間削減が得られるかを評価する必要があると考えている．

参考文献

[1] 日本応用数理学会 監修，櫻井 鉄也，松尾 宇泰，片桐 孝洋，"数値線形代数の数理と HPC"，共立出版，2018，p.p.1-2

[2] Shengen Yan, Chao Li, Yunquan Zhang, and Huiyang Zhou，"*YaSpMV: yet another SpMV framework on GPUs*"，SIGPLAN Not. 49, 8 (August 2014), 107–118，https://doi.org/10.1145/2692916.2555255

[3] Genshen Chu, Yuanjie He, Lingyu Dong, Zhezhao Ding, Dandan Chen, He Bai, Xuesong Wang, and Changjun Hu，"*Efficient Algorithm Design of Optimizing SpMV on GPU*"，In Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23)，Association for Computing Machinery, New York, NY, USA, 115–128, https://doi.org/10.1145/3588195.3593002

[4] Mickeal Verschoor, Andrei C. Jalba，"*Analysis and performance estimation of the Conjugate Gradient method on multiple GPUs*"，Parallel Comput. 38, 10–11 (October, 2012), 552–575.

[5] Weifeng Liu and Brian Vinter，"*CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication*"，In Proceedings of the 29th ACM on International Conference on Supercomputing (ICS '15), Association for Computing Machinery, New York, NY, USA, 339–350, https://doi.org/10.1145/2751205.2751209

[6] 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, France, 2013, pp. 11-20, doi: 10.1109/ICPP.2013.10.

[7] "NVIDIA A100"，https://www.nvidia.com/ja-jp/data-center/a100，閲覧日 (2024 年 1 月 8 日)

[8] "NVIDIA H100 Tensor Core GPU Architecture"，https://resources.nvidia.com/en-us-tensor-core，閲覧日 (2024 年 1 月 9 日)

[9] ”AMD EPYC 7302”，https://www.amd.com/en/product/8821，閲覧日 (2024年1月8日)

[10] ”NVIDIA Ampere architecture whiteparer jp”，https://www.nvidia.com/content/dam/en-zz/ja/Solutions/Data-Center/documents/nvidia-ampere-architecture-whitepaper-jp.pdf，閲覧日 (2024年1月8日)

[11] ジョン・L・ヘネシー，デイビッド・A・パターソン，”コンピューターアーキテクチャ定量的アプローチ [第6版]”，(中條 拓伯，天野 英晴，鈴木 貢) 訳，エスアイビー・アクセス，2019年，p.p.166-168

[12] ジョン・L・ヘネシー，デイビッド・A・パターソン，”コンピューターアーキテクチャ定量的アプローチ [第6版]”，(中條 拓伯，天野 英晴，鈴木 貢) 訳，エスアイビー・アクセス，2019年，p.168

[13] Hisa Ando，”GPUを支える技術”，技術評論社，2017年，p.p.153-155

[14] Ronald F. Boisvert，Roldan Pozo，Karin A. Remington，”*The Matrix Market Exchange Formats:Initial Design*”，U.S. Department of Commerce Technology Administration National Institute of Standards and Technology Computing and Applied Mathematics Laboratory Gaithersburg, MD 20899 USA, Decenber 1996.

[15] 日本応用数理学会 監修，櫻井 鉄也，松尾 宇泰，片桐 孝洋，”数値線形代数の数理とHPC”，共立出版，2018，p.p.278-282

[16] D. Merrill and M. Garland, ”*Merge-Based Parallel Sparse Matrix-Vector Multiplication*”, SC ’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, USA, 2016, pp. 678-689.

[17] NVIDIA Developer Forums, ”CUSPARSE implementation of SpMV”, https://forums.developer.nvidia.com/t/cusparse-implementation-of-spmv/217591，閲覧日 (2024年1月4日)

[18] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, and Alan R. Bishop，”*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 2014 36:5, C401-C423.

[19] E. Karimi, N. B. Agostini, S. Dong and D. Kaeli，”*VCSR: An Efficient GPU Memory-Aware Sparse Format*”, in IEEE Transactions on Parallel and Distributed Systems, vol. 33, no. 12, pp. 3977-3989, 1 Dec. 2022

[20] M. Maggioni and T. Berger-Wolf, "*CoAdELL: Adaptivity and Compression for Improving Sparse Matrix-Vector Multiplication on GPUs*" 2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, 2014, pp. 933-940

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

[22] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick and J. Demmel, "*Optimization of sparse matrix-vector multiplication on emerging multicore platforms*" SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing, Reno, NV, USA, 2007, pp. 1-12

[23] Kolodziej et al., *The SuiteSparse Matrix Collection Website Interface*, 2019, Journal of Open Source Software, 4(35), 1244

[24] Timothy A. Davis and Yifan Hu, "*The University of Florida sparse matrix collection*", ACM Trans. Math. Softw. 38, 1, Article 1 November 2011, p.25

研究業績

1. 村上 舜，米田 一徳，岩村 尚，渡邉 正宏，井口 寧，”GPU における疎行列密ベクトル積の高速化のための非ゼロ要素位置辞書圧縮を適用した疎行列格納形式の提案”，情報処理学会 第 190 回ハイパフォーマンスコンピューティング研究発表会，Aug．2023

2. 村上 舜，米田 一徳，岩村 尚，渡邉 正宏，井口 寧，”疎行列密ベクトル積の高速化のための非ゼロ要素位置辞書圧縮を適用した疎行列格納形式の GPU における形式変換の評価”，情報処理学会 コンピューターシステム研究会，Dec．2023

謝辞

本研究を進めるにあたり，多大な助言やご指導をいただいた井口 寧教授に深謝いたします．加えて，中間審査で有益なご助言をいただきました田中 清史教授，金子 峰雄教授，本郷 研太教授に感謝いたします．また，共同研究において疎行列データの提供，ご助言をいただきました富士通 Japan 株式会社の米田 一徳様，渡邉 正宏様，岩村 尚様に感謝いたします．最後に井口研究室の皆様には，ゼミでの議論や修士生活において大きな刺激をいただけたことに感謝の意を表します．