情報処理学会論文誌　数理モデル化と応用　Vol.13　No.2　93–106 (Aug. 2020)

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

河村 知記[1,a)]　米田 一徳[2]　岩村 尚[2]　渡邉 正宏[2]　井口 寧[1]

受付日 2020年1月31日，再受付日 2020年4月2日，
採録日 2020年5月15日

概要：近年，計算機の高性能化にともない数値シミュレーションをオンサイトでリアルタイムに実行し，様々な産業に応用することが期待されている．このようなシミュレーションは，GPGPU を利用することによって実用的な計算時間での実行が期待できるが，メモリ容量の制約が大きな問題点である．そこで本稿では数値シミュレーションの代表的手法である Finite Element Method（FEM）で現れる疎行列のメモリ使用量削減手法を提案する．提案手法では，疎行列の列番号を表す値をパッキングし，メモリに格納する値の数を削減する．複数の疎行列による評価では，15 個中 13 個において従来手法に対しメモリ使用量を削減し，最大で 26.3%の削減率となった．また，オンサイト実施が期待される分野の一例として心臓シミュレーション用の疎行列にも適用したところ，メモリ使用量が 20.6%削減された．

キーワード：GPGPU，SpMV，FEM，疎行列格納方式，圧縮率向上手法

A New Compression Scheme of Sparse Matrix Formats for Accurate Numerical Simulation on Site Environment with GPGPU

Tomoki Kawamura[1,a)]　Kazunori Yoneda[2]　Takashi Iwamura[2]
Masahiro Watanabe[2]　Yasushi Inoguchi[1]

Received: January 31, 2020, Revised: April 2, 2020,
Accepted: May 15, 2020

Abstract: Recent years, performing high-precision numerical simulations on-site and adapting the results to various industries are expected, as the computing power has been increasing. Although such simulations can possibly be executed in a practical time with GPUs, some of them do not fit in GPUs due to their limited memory capacity. To solve this problem, we proposed a new compression scheme of sparse matrix storage formats. Assuming that some parts of the column indices in the sparse matrix are a consecutive and such those parts can be described with its minimum and maximum column number. In our experiments, we reduced the memory usage of general sparse matrices up to 26.3% in 13 out of 15 matrices compared with conventional storage. Also, we reduced the memory usage of matrices used in a heart simulation up to 20.6%.

Keywords: GPGPU, SpMV, FEM, sparse matrix formats, compression method

1. はじめに

近年，様々な分野において高精度な数値シミュレーションの需要が増大している．通常，高精度な数値シミュレーションには PC クラスタやスーパコンピュータ等の大規模な計算機を使用する場合が多い．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増えている．たとえば医療分野における数値シミュレーション応用が考えられる．この分野では HeartFlow 社が $\mathrm{FFR_{CT}}$ [1] の商用サービスを始めているが，これは Amazon Web Services（AWS）を活用したクラウドサービスである [2].

[1] 北陸先端科学技術大学院大学
JAIST, Nomi, Ishikawa 923–1292, Japan
[2] 富士通株式会社
Fujitsu Limited, Kawasaki, Kanagawa 211–8588, Japan
[a)] t-kawamura@jaist.ac.jp

たとえば，狭心症が疑われる患者の CT 画像と診断情報が病院から AWS 上に構築されたサービス基盤に送信され，Heartflow の技術者が有限要素法による 3 次元の流体解析を実行，Fractional Flow Reserve（FFR）値を算出し，病院にレポートを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．これはカテーテルを用いる侵襲的な検査であるが，$\mathrm{FFR}_{\mathrm{CT}}$ 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 値を算出している．一方で，病院側からすると情報管理の観点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り，オンサイトのシミュレーション実行が望まれるケースがある．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，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，高精度かつオンサイトでの実行が可能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．従来高精度な数値シミュレーションを実用的な時間で実行するためには，大規模計算機が必要であったが，GPU を汎用的な計算に適用する技術の発展により，数値シミュレーションを省スペースかつ高速に実行可能となった．しかしながら，利用可能なメモリ量が，CPU は 1 ノードで数 TB までなのに対して，GPGPU では高々数十 GB しかないという大きな制約がある．数値シミュレーションの代表的な手法である Finite Element Method（FEM）を用いた場合，シミュレーションの大規模化，高精度化により，FEM で扱う疎行列のサイズが増大する．この疎行列が GPU メモリへ格納しきれない場合，CPU と GPU 間の転送が頻発し，演算性能が大きく低下する．

そこで本稿では，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圧縮手法を提案し，GPGPU の小容量のメモリに格納する．評価としていくつかの 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）[5] の行列を用いた．GPU はメモリ参照の特性や並列演算器の動作が CPU とは異なるため，圧縮した疎行列の演算速度が重要なポイントになるが，提案手法では GPGPU による演算速度は従来手法と遜色ないながらも，圧縮率を最大 26.3%向上できる．

本稿は，2 章で FEM における疎行列の説明と行列圧縮に関する関連研究を俯瞰する．3 章で提案する疎行列格納方式に対する圧縮方法を詳しく述べる．4 章で提案圧縮方法を評価し，5 章でまとめを行う．

2. FEM における疎行列とその圧縮法

2.1 FEM における疎行列

FEM は偏微分方程式を離散的に解く手法の 1 つである．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．これらの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に帰着する．

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．高精度な解析のためには有限要素をより細かくする必要があるが，それにともない疎行列のサイズも増大する．

大規模な疎行列の連立一次方程式を解く方法として，Algorithm 1 に示す Generalized minimum residual（GMRES）[6] のような反復法がよく用いられる．FEM によるシミュレーションの高速化につながることから，反復法の高速化に関する研究が数多く行われている [7], [8], [9].

Algorithm 1 は反復法の一例として GMRES のアルゴリズムを示している．アルゴリズム中，二重線の四角で囲まれた疎行列 A と，ベクトルの積を求める処理があり，これは Sparse matrix-vector multiplication（SpMV）と呼ばれる．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，非常に多くの演算時間が必要となる．SpMV は反復ごとに実

Algorithm 1 GMRES

1: Compute $r_0 = b - \boxed{\mathrm{A}}x_0$, $\beta := |r_0|_2$, 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** $j = 1, 2, \ldots, m$ **do**
4: Compute $\omega_j := \boxed{\mathrm{A}}v_j$
5: **for** $i = 1, \ldots, j$ **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 $x_m = x_0 + [v_1, \ldots, v_m]y_m$

行されるため，GMRES のボトルネックとなる．また，巨大な疎行列 A を格納するために多くのメモリを使用する．このことから，GPGPU における効率的な SpMV の高速化，省メモリ化のため，メモリへの疎行列格納方法が，多く検討されてきた [10]．

2.2　疎行列の格納方式

疎行列を格納する方式により，疎行列格納のためのメモリ使用量や SpMV の演算性能は大きく変化する．現在までに提案された様々な疎行列格納方式は，大きく分けると Compressed Sparse Row（CSR）と ELLPACK（ELL）[11] の 2 つが元となっている．**図 1** に CSR および ELL の格納方法を示す．CSR の長所は，無駄なゼロ要素をいっさい格納しないため，少ないメモリ使用量で疎行列を格納できる点があげられる．CSR では，3 つの配列を用いて疎行列を表す．1 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．2 つ目の配列は（図中 CSR の四角内，$Columns$），疎行列内の各非ゼロ要素の列番号を格納する．3 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 (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．

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

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 8 byte（64 bit）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 4 byte（32 bit）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 $8N_{\mathrm{z}}$，$4N_{\mathrm{z}}$ [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．

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，warp と呼ばれる演算単位が存在する．同一 warp 内のスレッドはつねに同じ処理を同時に行う．そのため，各スレッドが不連続なメモリアドレスにアクセスする場合は，スレッド数と同じ回数の逐次的なメモリアクセスが発生し，大きなオーバヘッドとなる．しかし，各スレッドが連続したメモリアドレスにアクセスする場合は，1 回のメモリアクセスですべてのスレッドが必要とするデータにアクセスすることが可能である．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．一般に疎行列の各行の非ゼロ要素数はばらつきがあることから，CSR に対してコアレスドアクセスをしようとすると，あるスレッドが別のスレッドが担当する行の値を読み込んでしまう等の問題が起きる．そのため，正しく SpMV の演算を行う際にはメモリアクセスが増加し，演算性能が低下する [12]．

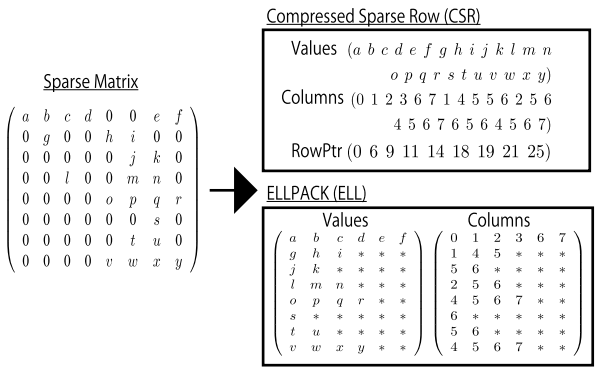


図 1　従来の疎行列格納方式

Fig. 1　Conventional storage formats for sparse matrices.

一方 ELL では，2 つの行列を用いて，疎行列を表す．1 つ目の行列（図 1 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．2 つ目の行列（図 1 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 2 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，2 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 1 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 内のスレッドがつねに連続したアドレスへアクセス可能である．したがって，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 [12]．

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，ELLPACK-R（ELL-R）を提案している [13]．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 (2)，(3) はそれぞれ，ELL と ELL-R を用いて疎行列を格納したときに必要となるメモリ使用量である．

$$MemUsage_{ELL} = 8NK + 4NK\ [\mathrm{byte}] \quad (2)$$

$$MemUsage_{ELL-R} = 8NK + 4NK + 4N\ [\mathrm{byte}] \quad (3)$$

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 と 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 (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) では第 3 項として $4N$ が追加される．

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．Monakov 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 [14]．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．しかしながら，SELL の列番号を保持する行列には，さらに削減可能なデータが存在する．SELL の問題を解決するため，Choi らは，Blocked CSR（BCSR）と Blocked ELL（BELL）を提案している [15]．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．規則にのっとり，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．BCSR と BELL 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．

Pinar 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 [16]．**図 2** 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，$Colind$ 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，Pinar らの研究では，GPU を始めとする並列計算環境における 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．たとえば，単純に GPU 上の 1 スレッドが図 2 の行列の 1 行を担当し，並列化を試みるとする．各 GPU 上のスレッドはまず，$Rowptr$ 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 BLOCK 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．たとえば，図 2 の行列の 5 行目（最初の行を 0 行目とする）の k を計算する場合，k の列番号は分かるが，スレッドは並列に動いているために k に対応する $Nzptr$ の数値が分からず，$Nzptr$ 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．

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

$$A_f = (a,b,c,d,e,f,g,h,i,j,k,l)$$
$$Colind = (0,1,3,1,2,1,4)$$
$$Rowptr = (0,1,3,4,5,7)$$
$$Nzptr = (0,3,4,6,8,10,11,12)$$

図 2　Blocked Compressed Row Storage（BCRS）を用いた格納例
Fig. 2　An example of using Blocked Compressed Row Storage (BCRS).

また，著者らもこれまでの研究 [17] 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，GPU メモリ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 の演算性能が低下することも明らかになった [17]．そこで本研究では，疎行列に必要なメモリ使用量の削減を第一に考えながら，非ゼロ要素の連続部と不連続部を分けることにより，メモリアクセス効率の向上を図り，圧縮による CSR，ELL を用いた SpMV 演算性能に対する悪影響が少ない，Row Block Packing Method（RBP 法）を提案する．メモリ使用量削減の方法として，単精度浮動小数点数の使用等の方法も考えられるが，本研究では非ゼロ要素の値ではなく，整数で表現される列番号数の削減を行う．そのため，本研究で提案する RBP 法を適用後，非ゼロ要素の値に対し，単精度浮動小数点数の適用等によりさらにメモリ使用量の削減を行うことも可能であると考えられる．

3. ROW BLOCK PACKING METHOD（RBP 法）

3.1 RBP 法の説明

2.1 節で述べた FEM における疎行列の非ゼロ要素のパターンは，節点間の接続関係と，節点番号により決定される．1 つの節点は近隣の節点と隣接していることから，疎行列内でも複数の連続した非ゼロ要素が存在すると考えられる．この疎行列内の連続する非ゼロ要素の列番号を圧縮することで，疎行列のメモリ使用量を削減可能である．FEM には非構造格子，構造格子の 2 つのデータ構造が存在するが，どちらも節点の繋がりが多いため，疎行列内に連続した非ゼロ要素が存在すると考えられる．そのため，

本提案手法は両データ構造を対象とする．

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．しかしながら，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すべての列番号を記憶する．そのため，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．少量の GPU メモリを効率的に使用することで，オンサイト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．

図 3 に RBP 法の概略図，Algorithm 2 に RBP 法を既存の疎行列格納方式に適用する手順を示す．Algorithm 2 では，疎行列内の非ゼロ要素を先頭から順に，図 3 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 3 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．

まず 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 3 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 BLOCK として考え，最終的に RBP 法を適用する疎行列格納方式へ BLOCK 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（2，3 行目）．

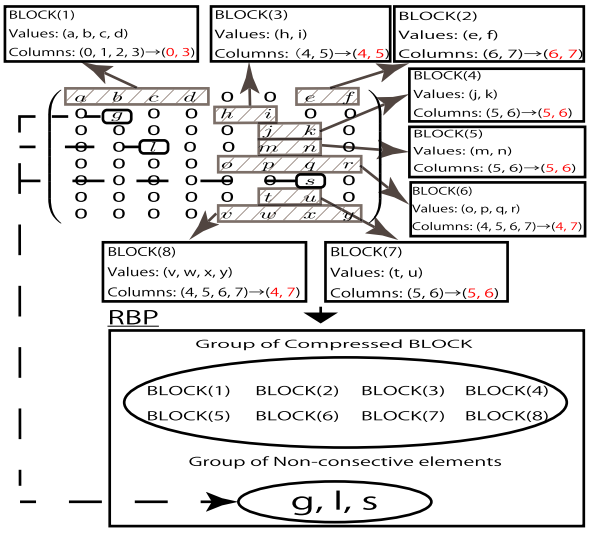


図 3　Row Block Packing（RBP）法の概略図
Fig. 3　Row Block Packing (RBP) method.

Algorithm 2 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
```

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 BLOCK 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 3 中，各 BLOCK 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かったりする場合，大きなメモリ使用量の削減が可能である．この結果，どれだけ長く連続した非ゼロ要素の場合でも，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かったりする場合，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，BLOCK 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 BLOCK を本来の 1 つの非ゼロ要素と同等に扱い格納する（*Values* と *Columns* の複数の要素を 1 つの塊として考える）．この過程は，CSR や ELL 等の既存の格納方式で値や列番号等を格納する方法を BLOCK 化された非ゼロ要素に対して適用することを意味する．たとえば CSR であれば，BLOCK 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．このとき，ELL 等の 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 内の各スレッドはつねにコアレスドアクセスを行うことが可能である．

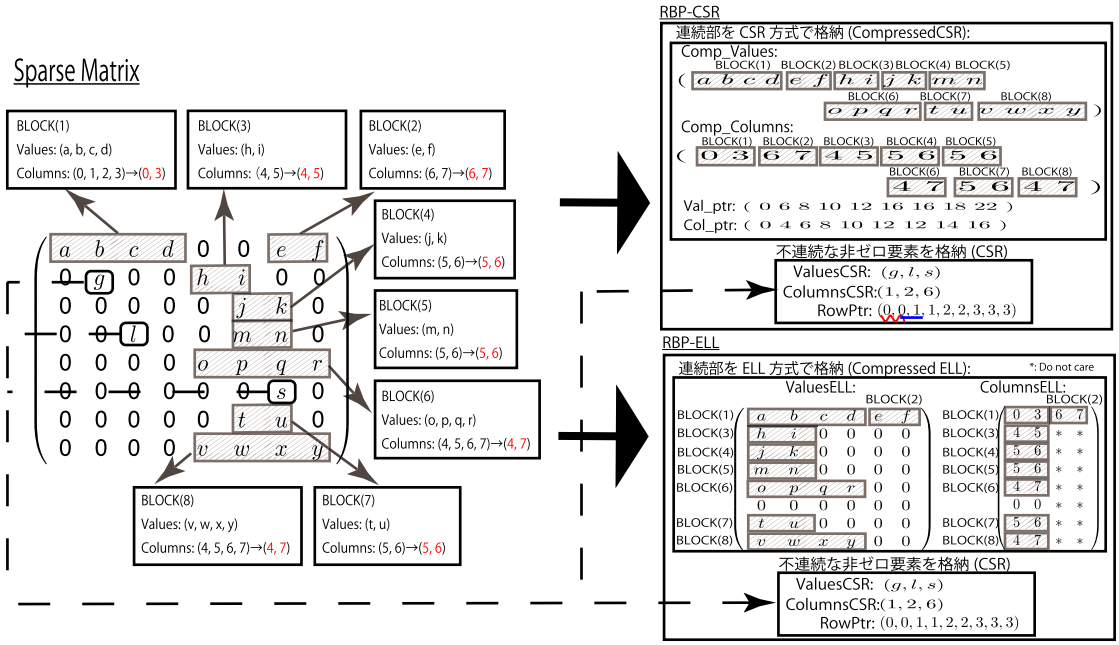


図 4　RBP-CSR と RBP-ELL への変換
Fig. 4　Conversion to RBP-CSR and RBP-ELL.

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 の格納方式にのっとった BLOCK の格納方法を図 4 を用い，次の節から説明を行う．

RBP 法は複数の値を BLOCK 化するというシンプルな方法をとっているため，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提案圧縮方法を適用可能であると考えられる．たとえば，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．

3.2　CSR への RBP 法の適用（RBP-CSR）

図 4 に RBP 法を施した CSR（RBP-CSR）の概要を示す．RBP-CSR は Algorithm 3 にのっとり，CSR の列番号を格納する配列の要素数を削減していく．Algorithm 3 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，Algorithm 2 に比べ複雑に見えるが，同じ流れで処理を行っている．Algorithm 3 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 $RowPtr[i+1]-1$ 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．Algorithm 3 中の Col_{CSR}，Val_{CSR}，$RowPtr_{CSR}$ は図 1 の CSR の配列に対応している．また，その他の配列は図 4 に対応している．

Algorithm 3 の 6 行目から 23 行目までが Algorithm 2 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 1 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（Algorithm 2 の 3 行目に対応）．While 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（Algorithm 2 の 4 から 6 行目に対応）．たとえば，図 4 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．

Algorithm 2 の 8 から 10 行目までの不連続部の要素に対する処理は，Algorithm 3 の 24 行目から 29 行目に対応する．$TmpCol$ に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 4 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．

Algorithm 3 の 30 行目から 40 行目は，調査している非ゼロ要素と 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，Algorithm 2 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 4 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，0 行目（最初の行を 0 行目とする）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

Algorithm 3 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
```

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 1 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．Algorithm 3 の 41 行目で，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，連続した非ゼロ要素がまったく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 4 の行列では 0 行目には不連続な非ゼロ要素が存在せず，1 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 4 中，赤い波線）を格納する．1 行目は $RowPtr$ の 2 番目と 3 番目 “0，1”（図 4 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 Algorithm 3 を図 4 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 4 中の RBP-CSR を得ることが可能である．

式 (4) 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 4 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 (4) 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．

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

Algorithm 4 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 Algorithm 4 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．Algorithm 4 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 $Comp\_Values$ から読み込むため，Val_ptr から担当する行に対応するインデックスを読み込む．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 ループで，$ColNow$ 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

Algorithm 4 SpMV code of RBP-CSR on GPU

```
 1: Let id be thread ID (id: 0 to n−1)
 2: Set Count, TmpResult = 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+ = ValuesCSR[j]∗
       dVector[ColmunsCSR[j]]
18: end for
19: Result[id] = TmpResult
```

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 の計算を行う．

14 行目から 18 行目は，不連続な非ゼロ要素を計算する部分である．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．

3.3　ELLPACK への RBP 法の適用（RBP-ELL）

図 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，Algorithm 2 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，BLOCK の $Values$ と $Columns$ のデータを ELL の方式にのっとり，$ValuesELL$ と $ColumnsELL$ にインデックスの小さい BLOCK から格納していく．ここでは図 4 の Compressed ELL のように，各 BLOCK 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．RBP-ELL においても不連続な非ゼロ要素を CSR 形式で格納することで，パディングの数を抑え，メモリ使用量増加を防ぐ狙いがある．

Algorithm 5 SpMV code of RBP-ELL on GPU

```
 1: Let id be thread ID (id: 0 to n−1)
 2: Set Count, TmpResult = 0
    /* K_v is number of columns in ValuesELL */
 3: Set K_v
    /* 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 K_v − 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
          /* K_c is number of columns in ColumnsELL */
14:       if Count > K_c 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
```

式 (5) 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．

$$MemUsage_{RBP-ELL} = 8NK_{\mathrm{v}} + 4NK_{\mathrm{c}} + 8N_{\mathrm{non}} + 4N_{\mathrm{non}} + 4(N+1)\ [\mathrm{byte}] \tag{5}$$

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 $ValuesELL$，$ColumnsELL$ が必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である．$8N_{\mathrm{non}} + 4N_{\mathrm{non}} + 4(N+1)$ は図 4 中 RBP-ELL の $ValuesCSR$，$ColumnsCSR$，$RowPtr$ が必要とするメモリ使用量を示す．

Algorithm 5 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している．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 K_v-1 まで j を増加させていく．K_v は RBP-ELL の

ValuesELL 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 行目で RBP-CSR のとき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 行目で，$StartELL$ 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 行目から 16 行目の処理は，スレッドが担当している行のすべ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処理である．そして，19 行目から 23 行目では，図 4 の不連続な非ゼロ要素を格納してある CSR を用いて SpMV 計算を行う．その後，連続部の計算結果が格納されている $TmpResult$ に不連続分の計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．

4. RBP 法の性能評価実験

本章では，提案した圧縮方法，RBP 法の性能を評価する．RBP 法を用いた場合の疎行列格納方式のメモリ使用量，CPU–GPU 間のデータ転送時間，そして SpMV の演算時間に対する影響を評価する．

4.1 評価実験環境

表 1 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 2** に示す．最初の 13 個は，従来手法である CSR や ELL と比較するため Florida Sparse Matrix Collection [18] 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 6 GB，同じく 2014 年では K40 が 12 GB 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 [5] 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（rma10，UT-Heart1，UT-Heart2 以外の疎行列は，収録されている上三角または下三角行列のみを使用）．また，表 2 中の N，N_z，N_{non}，K，K_{v}，K_{c} は，式 (1) から (5) で使用されているものに対応している．本実験で使用する疎行列格納方式は，CSR，RBP-CSR，ELL，RBP-ELL に加え，ELL-R と ELL-R に RBP 法を適用した RBP-ELL-R である．RBP-ELL-R の説明は誌面の都合上省略したが，ELL と同様の方法で RBP 法を適用可能である．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．

図 5 は実験で使用する，FEM で生成された連立一次方程式（疎行列）に RBP 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 中の赤枠，つまり疎行列ファイルの読み込みから 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 RBP 法の適用，GMRES の初期処理までを CPU 上で処理し，その後は GPU 上での処理となる．図 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()”

表 1　評価実験に使用した環境
Table 1　Environment for the experiments.

	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

表 2　評価実験に用いた疎行列
Table 2　Various sparse matrices for the experiments.

Name of matrix	N	N_{z}	N_{non}	K	K_{v}	K_{c}
cant	62,451	4,007,383	39,097	40	40	16
rma10	46,835	2,374,001	447	145	145	40
consph	83,334	6,010,480	30,461	78	78	38
parabolic_fem	525,825	3,674,625	999,115	7	6	6
pwtk	217,918	11,524,432	199	90	90	18
thermal2	1,228,045	8,580,313	2,846,134	11	9	8
af_shell9	504,855	17,588,845	919	30	30	10
F1	343,791	26,837,113	46,738	378	378	150
nd24k	72,000	28,715,634	441,476	483	472	114
dielFilterV2real	1,157,456	48,538,952	10,631,364	99	79	48
Cube_Coup_dt0	2,164,760	124,406,070	125,891	52	52	23
Bump_2911	2,911,419	127,729,899	212,859	174	174	44
Queen_4147	4,147,110	316,548,962	228679	63	63	35
UT-Heart1	82,047	3,423,519	0	63	63	40
UT-Heart2	130,595	6,954,413	969,595	124	111	50

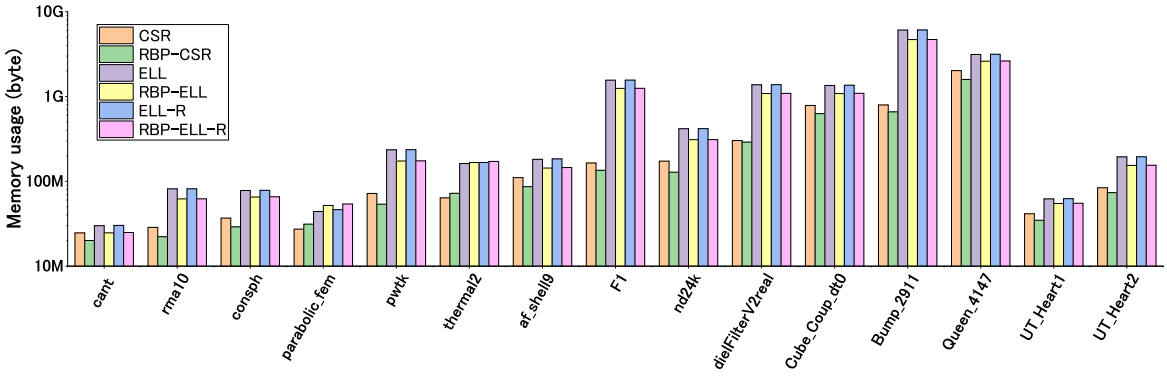


図 6　各疎行列格納方式のメモリ使用量

Fig. 6　Memory usage of each storage formats.

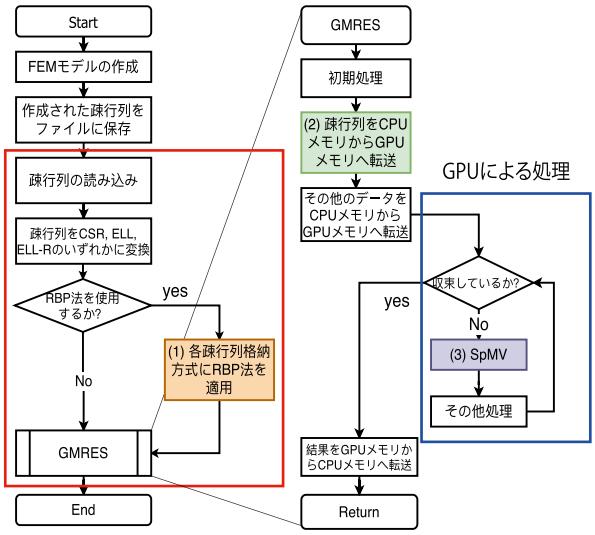


図 5　RBP 法を用いた GMRES の計算モデル

Fig. 5　Model of calculating GMRES with RBP method.

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 の処理すべてを CPU 上で行う．

4.2 節では，各疎行列格納方式で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，4.3 節では各疎行列格納方式を用いて図 5 内の SpMV（赤い四角部分）の SpMV を 1 回実行した際の演算時間を評価する．最後に 4.4 節で，図 5 内の赤枠全体の演算時間の評価を行う．

4.2　各疎行列格納方式のメモリ使用量の評価

図 6 に，各疎行列格納方式を用いて，表 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 (1) から (5) 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 (6) 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) に $4N$ を追加した形となっている．

$$MemUsage_{RBP-ELL-R} = 8NK_{\mathrm{v}}+4(NK_{\mathrm{c}}+N)+8N_{\mathrm{non}} +4N_{\mathrm{non}}+4(N+1)\ [\mathrm{byte}] \quad (6)$$

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，それぞれ表 2 の 15 個中 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 では nd24k において，最大の 26.0%のメモリ使用量の削減に成功した．15 個の疎行列における平均のメモリ使用量の削減率は，14.5%であった．この結果から，FEM で生成される行列には多くの連続した非ゼロ要素が存在することが確認できる．RBP-ELL では，pwtk において，メモリ使用量の削減率が高く，ELL に比べ，26.3%の削減に成功している．また，RBP-ELL の 15 個の疎行列の平均メモリ使用量削減率は 16.2%となった．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 行列で，削減率は 26.2%であった．平均メモリ使用量削減率は，16.2%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 6 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 のメモリ使用量は CSR に比べ 12.4%，RBP-ELL のメモリ使用量は ELL と比べ 20.6%，RBP-ELL-R のメモリ使用量は ELL-R と比べ 20.6%少ない結果となった．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．またさら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 6 GB 以下に削減することに成功した．これにより GeForce 1060 の 6 GB 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

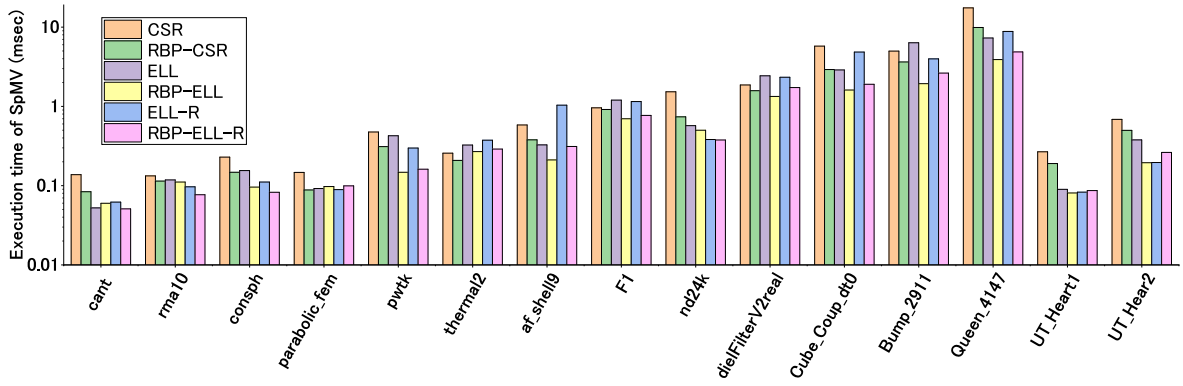


図 7　GPU における SpMV の演算時間
Fig. 7　Execution time of SpMV on GPU.

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，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ていない．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 や thermal2 では，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．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．しかし，今回は CSR や ELL と比較するため，本稿では再割当てについて言及しない．これらの要因をふまえ，RBP 法を既存の疎行列格納方式に適用した際に，使用メモリ量を削減可能な条件を式 (7)，(8) に示す．式 (7) は RBP-CSR が使用メモリ量を削減可能な条件，式 (8) は RBP-ELL，RBP-ELL-R が使用メモリ量を削減可能な条件である．式中で使用される変数に関しては，式 (1) から (6) までに使用している変数と同じである．それぞれの式の左辺は RBP 法により削減されるメモリ量を表している．右辺は RBP 法を適用することにより，追加される CSR 等の配列に必要となるメモリ量を表す．2 つの式から，疎行列内に多くの不連続な非ゼロ要素が存在する場合には，多くのメモリ量を削減する必要があることが分かる．

$$2(N_{\mathrm{z}} - N_{\mathrm{val}}) + (N_{\mathrm{z}} - N_{\mathrm{col}}) > 3N_{\mathrm{non}} + 2(N+1) \quad (7)$$

$$N(3K - 2K_{\mathrm{v}} - K_{\mathrm{c}}) > 3N_{\mathrm{non}} + (N+1) \quad (8)$$

4.3　GPU 上での SpMV 演算時間の評価

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 7 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．RBP-CSR ではすべての疎行列で，CSR に比べ SpMV 演算時間が短縮された．RBP-ELL では 15 個中 13 個，RBP-ELL-R では 15 個中 12 個で既存手法より SpMV 演算時間が短くなった．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 2.07 倍の高速化を達成した．また平均では，1.50 倍の高速化となった．RBP-ELL は，Bump_2911 において最大の 3.28 倍の高速化に成功している．全体の平均高速化率は 1.65 倍である．RBP-ELL-R において，最も高速化に成功した行列は af_shell9 であり，3.32 倍の高速化に成功している．平均高速化率は 1.51 倍である．この結果から，圧縮率が高い疎行列においては RBP 法は，SpMV 演算性能向上のための最適化技術の 1 つとして使用可能である．既存の疎行列格納方式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，SpMV カーネルプログラム（Algorithm 4，5）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と推測する．一方 RBP-ELL，RBP-ELL-R では，使用メモリ量を削減できなかった疎行列（parabolic_fem，thermal2）では，メモリアクセス数の削減が達成されず，RBP 法で追加される配列へのメモリアクセスが増加したことから，RBP 法を使用した格納方式の SpMV 演算時間は，増加している．また RBP-ELL-R では，UT_Heart1，UT_Heart2 において SpMV 演算時間が ELL-R に比べ長くなっている．これは圧縮により，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原因と考えられる．連続した非ゼロ要素の数がそれぞれの

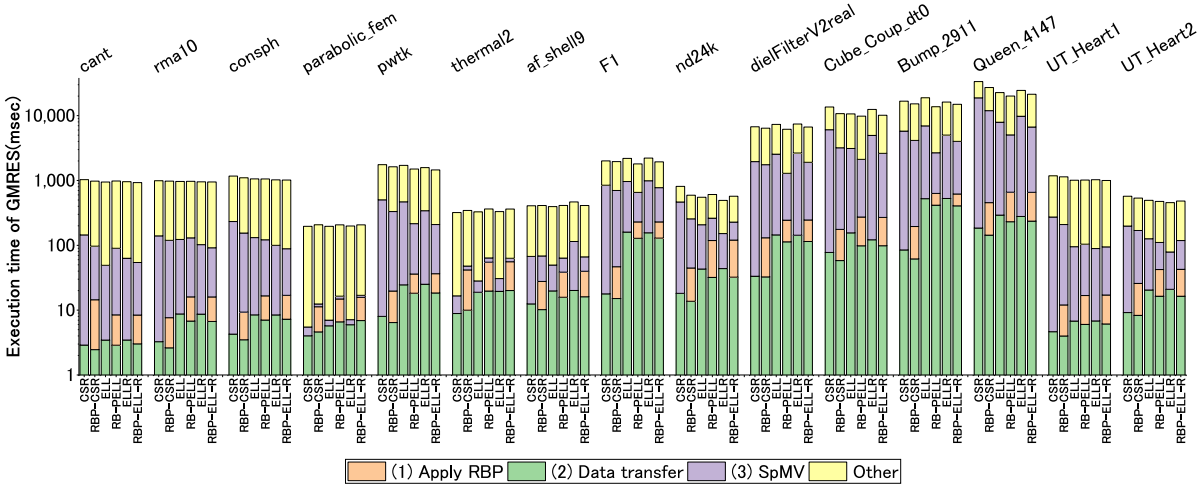


図 8　GPU における GMRES の演算時間
Fig. 8　Execution time of GMRES on GPU.

BLOCK で異なることから，スレッドが 1 個の BLOCK を処理するために，ValuesELL がアクセスされる要素数は，スレッドごとに異なる．よって，連続したアドレスへのアクセスが困難となり，演算時間が増加したと考えられる．

UT-Heart1，UT-Heart2 では，RBP-ELL を使用することで．ELL-R と同等の SpMV 演算時間でありながらメモリ使用量は ELL-R 以下となった．このことから，さらなる高精度なシミュレーションをオンサイトで行うことが可能である．RBP 法は，メモリ使用量の少ない CSR をさらに省メモリ化したり，SpMV を高速に演算可能な ELL や ELL-R のメモリ使用量を削減したりすることで，使用する GPU のメモリ量を考慮した疎行列格納方式の選択の幅を広げる．

4.4　GMRES による評価実験結果

図 8 に，疎行列読み込みから連立一次方程式求解全体の処理時間を評価するため，Algorithm 1 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 における (1) CPU 上で RBP 法を適用するための処理時間，(2) 疎行列の CPU メモリから GPU メモリへの転送時間，(3) GPU 上での SpMV 演算時間の合計，およびその他の処理時間を，それぞれオレンジ，緑，紫，黄色で表した．その他の処理時間は，GMRES（図 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．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かか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，必要とする精度等により必要な反復回数は問題により異なるため，最大 1,000 回としており，1,000 回で収束しない疎行列に関しては 1,000 回反復した際の処理時間を記載している．parabolic_fem，thermal2，af_shell9，F1，nd24k，UT_Heart2 はそれぞれ 12，26，86，649，283，277 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．

まず RBP 法を適用するための処理時間に着目すると，SpMV 1 回の演算時間に比べ，長い処理時間が必要となっている．RBP 法の適用は CPU で行われているため，SpMV の処理に比べ低速である．しかしながら，RBP 法の適用は 1 回のみであり，SpMV の実行は反復ごとに行われるため，SpMV 演算時間の合計に対する割合では図 8 から多くの疎行列において RBP 適用時間より，SpMV 演算時間が支配的となっていることが確認できる．GMRES 全体の演算時間を既存格納方式と RBP 法を適用した格納方式で比較すると平均で 6.6%，4.1%，3.5%と RBP 法を適用した場合の方が短い結果となった．反復回数が最も少ないほど RBP 適用のオーバヘッドは支配的になるが，反復回数が少ない parabolic_fem，thermal2，af_shell9 においても，既存手法に比べ 10%以下の演算時間増加に止まっている．反復回数が 1,000 回の疎行列においては，RBP 法を用いた場合でも全体処理時間が同等または短い結果となっている．反復回数が多くなるほど RBP 法を適用するメリットが大きくなると考えられる．この結果から，GMRES 全体の演算時間としては RBP 法を用いて圧縮を行ったとしても，遜色ない演算時間でより高い圧縮率が得られることが確認された．

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，13.6%，13.1%，13.4%削減された．nd24k において，RBP-CSR は最大の 13.6%のデータ転送時間削減を達成した．pwtk において，RBP-ELL，RBP-ELL-R はそれぞれ 25.8%，26.6%のデータ転送時間削減に成功しており，全行列中で最大の削減率である．この結果から RBP 法によるメモリ使用量削減にともな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

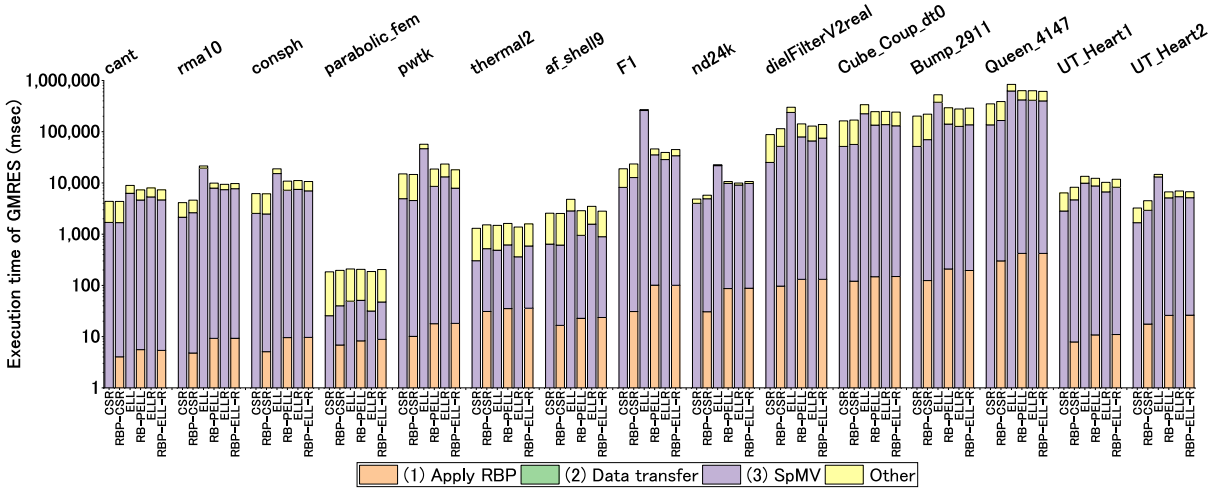


図 9　CPU における GMRES の演算時間
Fig. 9　Execution time of GMRES on CPU.

た parabolic_fem，thermal2 においては最大 15.8%のデータ転送時間の増加が確認された．

また，GPU による高速化の有効性を確認するため，すべての処理を CPU で行った場合の処理時間，**図 9** を記載する．縦軸は対数軸となっている．CPU での CSR，ELL，ELL-R，RBP-CSR，RBP-ELL，RBP-ELL-R と GPU での結果を比較すると，平均で 14.9，122.1，56.7，27.0，70.3，67.7 倍の高速化となった．反復法においてデータ転送は 1 回のみであるため，反復ごとに実行される SpMV の演算時間が支配的になる．図 8，図 9 から，CPU で支配的な SpMV 演算時間が GPU による高速化で大きく短縮されていることが分かる．そのためデータ転送時間を必要としても GPU を用いることは非常に効果的であり，GPU での利用範囲を広める RBP 法の有効性も示せていると考える．

5.　おわりに

本研究では，高精度な数値シミュレーションを GPGPU によりオンサイトで実現するための課題の 1 つとして，連立一次方程式の求解の省メモリ化に取り組み，FEM で生成される疎行列の特殊性を考慮した，疎行列格納方式に対する圧縮方式を提案した．本提案圧縮方法である RBP 法は，FEM が生成す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，連続した非ゼロ要素の列番号を圧縮し，格納する方式を採用している．メモリ使用量の評価実験では，RBP 法を用いることで CSR のメモリ使用量を平均 14.5%，ELL のメモリ使用量を平均 16.2%削減した．また GMRES の演算時間評価実験において RBP 法が，RBP 法の適用時間を考慮しても既存手法と遜色なく，メモリ使用量が少ない提案手法であることを示した．

今後の課題として，連続した非ゼロ要素が少ない疎行列においても，メモリ使用量の増加が起こらないよう，RBP 法を改良することがあげられる．

参考文献

[1] Min, J.K., Taylor, C.A., Achenbach, S.K., Kwon, B., Leipsic, J., Nørgaard, B.L., Pijls, N.J. and 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. and 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. and 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*), pp.1–10, ASME (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. and 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] Sugiura, S., Washio, T., Hatano, A., Okada, J., Watanabe, H. and 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).

[6] Saad, Y.: Iterative methods for sparse linear systems, second edition, *Society for Industrial and Applied Mathematics* (2003).

[7] Bahi, M.J., Couturier, R. and Khodja, Z.L.: Parallel GMRES implementation for solving sparse linear systems on GPU clusters, *Proc. 19th High Performance Computing Symposia*, pp.12–19 (2011).

[8] Chen, C. and Taha, M.T.: A communication reduction approach to iteratively solve large sparse linear systems on a GPGPU cluster, *Journal of Cluster Computing*, Vol.17, No.2, pp.327–337, Springer US (2013).

[9] DeVries, B., Iannelli, J., Trefftz, C., O'Hearn, A.C. and Wolffe, G.: Parallel implementations of FGMRES for solving large sparse non-symmetric linear systems, *Proc. 2013 International Conference on Computational Science*, Vol.18, pp.491–500 (2013).
[10] Filippone, S., Cardellini, V., Barbieri, V., et al.: Sparse Matrix-Vector Multiplication on GPGPUs, *ACM TOMS*, Vol.43, No.4, Article 30 (2017).
[11] Kincaid, R.D., Oppe, C.T. and Young, M.D.: ITPACKV 2D User's Guide, CNA-232 (1989), available from ⟨https://www.ma.utexas.edu/CNA/ITPACK/manuals/userv2d/node3.html⟩.
[12] Bell, N. and Garland, M.: Efficient sparse matrix-vector multiplication on CUDA, NVIDIA Technical Report NVR-2008-004, NVIDIA (2008), available from ⟨http://www.nvidia.com/docs/IO/66889/nvr-2008-004.pdf⟩.
[13] Vázquez, F., Fernández, J.J. and Garzón, M.E.: 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).
[14] Monakov, A., Lokhmotov, A. and Avetisyan, A.: Automatically tuning sparse matrix-vector multiplication for GPU architectures, *Proc. High Performance Embedded Architectures and Compilers* (*HiPEAC*), Vol.5952, pp.111–125 (2010).
[15] Choi, W.J., Singh, A. and Vuduc, W.R.: Model-driven autotuning of sparse matrix-vector multiply on GPUs, *Proc. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Vol.45, pp.115–126 (2010).
[16] Pinar, A. and Heath, T.M.: Improving performance of sparse matrix-vector multiplication, *Proc. 1999 ACM/IEEE Conference on Supercomputing*, pp.30–es (1999).
[17] Kawamura, T., Yoneda, K., Yamazaki, T., Iwamura, T., Watanabe, M. and Inoguchi, Y.: A compression method for storage formats of sparse matrix in solving the large-scale linear systems, *Proc. IEEE International Parallel and Distributed Processing Symposium Workshops* (*IPDPSW*), pp.924–931, APDCM (2017).
[18] Davis, A.T. and Hu, F.: The university of Florida sparse matrix collection, *ACM Trans. Math. Softw.*, Vol.38, Article 1 (2011).

河村 知記　（正会員）

1991 年生．2013 年富山高等専門学校制御情報システム工学科卒業．2015 年北陸先端科学技術大学院大学博士前期課程修了．GPGPU に関する研究に従事．

米田 一徳

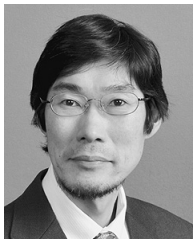
1984 年生．2007 年横浜国立大学工学部電子情報工学科卒業．2009 年同大学大学院修士課程修了．同年富士通株式会社に入社し，心臓シミュレータ共同開発プロジェクトにおいて大規模並列プログラムの高速化技術開発に従事．IEEE 会員．

岩村 尚

1980 年生．2003 年東京大学工学部化学システム工学科卒業．2006 年同大学大学院新領域創成科学研究科修士課程修了．同年富士通株式会社入社．2013 年東京大学大学院新領域創成科学研究科博士課程修了．科学博士．心臓シミュレータ共同開発プロジェクトにおいて冠循環シミュレーション技術開発に従事．2018 年文部科学大臣賞受賞．計算工学会会員．

渡邉 正宏

1971 年生．2004 年北陸先端科学技術大学院大学博士課程修了．情報学博士．2003 年文部科学省 IT プログラム VizGrid プロジェクトに研究員として参画．2007 年富士通株式会社に入社し，心臓シミュレータ共同開発プロジェクトにおいて可視化技術開発に従事．2018 年文部科学大臣賞受賞．日本シミュレーション学会理事．可視化情報学会代議員．



井口 寧　（正会員）

1991 年東北大学工学部機械工学科卒業．1994～1997 年日本学術振興会特別研究員．1997 年北陸先端科学技術大学院大学情報科学研究科博士後期課程修了．同大学情報科学センター助手，准教授を経て，現在，情報社会基盤研究センター情報環境研究開発部門教授．また，2002～2006 年まで科学技術振興機構さきがけ研究 21（機能と構成）に参加し研究に従事．2008～2009 年米国南フロリダ大学上級客員研究員．この間並列システム，ウェーハスタック集積システム，FPGA 等を用いた並列処理に関する研究を行う．IEEE 会員，電子情報通信学会シニア会員．本会シニア会員．