情報処理学会論文誌 プログラミング **Vol.7 No.1 10–17 (Jan. 2014)**

CUDAによるランダムスパース方程式求解の命令レベル並列性を用いた高速化手法

富永 浩文[1,a) 前川 仁孝[1]

受付日 2013年7月4日, 採録日 2013年10月9日

概要：本論文では，CUDA GPU において実非対称（ランダムスパース）な構造を持つ方程式の求解を高速化するために，スタティックスケジューリングによって抽出した命令レベル並列性を用いてベクトル命令を生成するアルゴリズムを提案する．従来より，ランダムスパース方程式求解には直接法が用いられている．直接法によるランダムスパース方程式の求解は，零要素を含む演算を省くことで効率良く実行できる．よって，CUDA GPU 上で演算の効率を高めるためには，スタティックスケジューリングを用いて零演算を含まない演算のみを抽出しベクトル化する．本手法は，方程式を求解する際に実行可能であるという組合せの情報のみを抽出することで，ベクトル化する．しかし，従来のように実行可能であるという組合せの情報のみからベクトル化するだけでは，高い効率で計算ができない．スケジューリングする際に，依存関係以外の情報も考慮することで CUDA GPU で演算の高速化が期待できる．そこで本論文では，すべての演算の依存関係を用いてスタティックスケジューリングを行うことで同時に実行可能な演算を抽出し，ベクトル化する．本手法によるスケジューリングでは，同時に実行可能な命令のうち，依存関係を多く持つ演算から優先的にベクトル化することで CUDA GPU による効率的な演算を可能にする．

キーワード：CUDA，直接法，近細粒度並列処理，スタティックスケジューリング

The Speed-up Method Solving Random-sparse Equations Using Instruction-level Parallelism by the CUDA

Hirobumi Tominaga[1,a) Yoshitaka Maekawa[1]

Received: July 4, 2013, Accepted: October 9, 2013

Abstract: This paper proposes a method to speed-up random-sparse equations solving using the CUDA GPU. The direct method has been used for solving random-sparse equations. In case of solving random sparse equations by the direct method, computation results of elements are zero during LU decomposition. In order to increase efficiency of operation, it is necessary to extract a lot of parallelizable operations without zero elements. The previous methods parallelize some operations which enable executions concurrently. However, it is able to speed-up more calculate, with consideration of the execution sequence of operations. Therefore, for computation faster, it is necessary to use instruction-level parallelism by static scheduling. This method is examining combination of parallel executable operations from dependencies among operations required for solving equations. For this purpose, it then processes in the CUDA GPU efficiently by extracting an instruction using the static scheduling and vectorization. The proposed method extracts concurrently executable instructions from dependency among operations. Then, the extracted instructions are converting into vector instructions, and the CUDA GPU processes them.

Keywords: CUDA, direct method, near finegrain parallel processing, static scheduling

1. はじめに

近年，SIMD 型ベクトルプロセッサである Graphics Processor Unit（GPU）を画像処理以外の分野において利用す

[1] 千葉工業大学情報工学科
Department of Computer Science, Chiba Institute of Technology, Narashino, Chiba 275–0016, Japan
[a) tominaga@mae.cs.it-chiba.ac.jp

© 2014 Information Processing Society of Japan

る Comupte Unified Device Architecuture（CUDA）[1] が注目されている．CUDA GPU は，多くの統合型プロセッサを搭載し広帯域なメモリバンド幅を持つメモリを搭載する高い演算性能を持つアーキテクチャである．数値計算の分野では，CUDA GPU を用いて性能を十分に引き出せるようにアルゴリズムを最適化することで，高い高速化を得ることが報告されている [2], [3]．このため，数値計算を扱う分野において高速化が困難であるといわれている電子回路の過渡解析や電力潮流計算など [4] に対しても高い高速化が期待できる．

電子回路や電力潮流計算は，非線形連立微分方程式を解くことで解析する．非線形連立微分方程式を求解するためには，数値積分法による差分化と Newton-Raphson 法による線形化を行い，生成された実非対称でランダムスパースな係数を持つ連立方程式（ランダムスパース方程式）の求解を繰り返し行う必要がある [4], [5], [6]．ランダムスパース方程式は，係数が帯などの特定の形にならず解が収束しないことがあるため，大規模問題でよく使われる反復法のアルゴリズムを用いて高速に求解することが困難である [6], [7]．このため，ランダムスパース方程式の求解には，ガウスの消去法や LU 分解法などの直接法が用いられる [6], [7]．直接法の求解を CUDA GPU を用いて高速化するためには，多くの並列化可能なデータを処理し実行効率を高める必要がある．

CUDA GPU を用いた直接法によるランダムスパース方程式求解手法として，行列をブロックに分割する手法 [8], [9] や列レベルのタスクスケジューリングを行う手法 [10], [11] が提案されている．行列をブロックに分割する手法は，ブロック内とブロック間の並列性を抽出し同時に複数のブロックを処理する．また，タスクスケジューリングを行う手法は，内積形式 LU 分解法が列ごとに計算の特徴を利用し，列どうしのデータ依存を解析して同時に実行可能な列どうしを組み合わせ，並列に実行する．これらの手法は，行列構造をなるべく密行列に近い形に変換し，密行列向けの求解手法を用いるため，スパース性を考慮した十分な並列性を抽出することが難しい．

ベクトルプロセッサにおいて，スパース性を考慮するために，命令レベル並列性を用いて高い並列性を抽出する手法として，拡張ベクトル化 LU 分解法が提案されている [4], [12]．本手法は，処理に必要なすべての命令のマシンコードを生成し，生成したコード間の依存関係を抽出して，依存関係から並列化可能な命令をベクトル化することで方程式の求解を高速化する．

CUDA GPU 上においても拡張ベクトル化 LU 分解法で生成したベクトル命令を用いることで必要最低限の演算のみを効率的に並列化し，ランダムスパース方程式求解を高速化できる考えられる．しかし，拡張ベクトル化 LU 分解法は実行レベルの情報のみを用いてベクトル化するため，同一レベルの除算演算と積差演算の命令を含んだ命令がベクトル化される可能性がある．このような命令を CUDA GPU で処理する場合，分岐処理のワープダイバージェントによる実行効率の低下が起こり，処理速度が低下する．一般的に，ワープダイバージェントは，分岐処理による演算回数が増加し演算効率に大きく影響を与える．この問題を解決するためには，従来よりアルゴリズムの改良によりワープダイバージェントを防止することが高い演算効率を得るうえで重要である．そこで，本論文では，ランダムスパース方程式を CUDA GPU で効率良く求解するために，拡張ベクトル化 LU 分解法を改良し同一演算命令を抽出してベクトル化することでワープダイバージェントを防止する手法を提案する．

2. 命令レベル並列性を用いた拡張ベクトル化 LU 分解法

拡張ベクトル化 LU 分解法は，LU 分解法全体の並列性を抽出する Maximum Vectorized Algorithm（MVA）法 [12] を改良した手法である．LU 分解法の分解処理と代入計算に必要な演算は式 (1) で示す除算演算と積差演算の 2 種類の計算式に帰着できる．このため，拡張ベクトル化 LU 分解法は 2 種類の演算命令をベクトル化したマシンコードを生成する．

$$\begin{cases} a = a/b \\ a = a - b \times c \end{cases} \tag{1}$$

拡張ベクトル化 LU 分解法は，生成される各命令に対して実行レベルを設定することで並列化可能な命令の組合せを決定する．**図 1**，**図 2** に除算演算と積差演算のレベル付けの手順を示し，**図 3** に抽出した実行レベルを用いてベクトル化する手順を示す．図中の要素番号は係数行列の要素番号であり，コード番号は方程式を求解するために必要な演算命令を格納する順序である．レベル付けの手順は，まず，演算に用いる要素の実行レベルが格納されている参照要素のレベル付け配列を比較し，大きい方の実行レベルに +1 した値をこの演算の実行レベルとして実行レベルのレベル付け配列にコード番号の低い順から格納する．次に，自身を更新するために更新する要素の参照要素のレベル付け配列にも決定したレベルを設定する．図 2 の例では，6 番目の要素に対して積差演算する．このため，4 と 2 の要素を参照しながら計算する．本例では 3 つの実行レベルで一番最大の実行レベル $1 + 1 = 2$ を自身の実行レベルとして，レベル付け格納配列に格納する．すべての演算の実行レベルを実行レベルのレベル付け配列に格納したら，図 3 に示すように同一の実行レベルの演算の命令を小さい順から集めベクトル命令を生成する．以下に，実行レベルの決定とベクトル化の手順を示す．

(1) 係数行列に対応する各要素の演算レベルを格納する

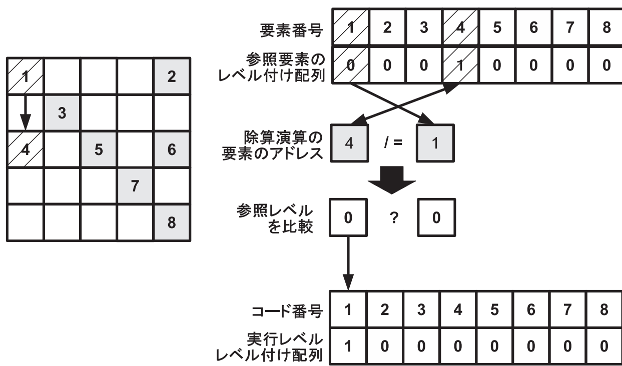

図 1　拡張ベクトル化 LU 分解法による除算演算のレベル付け
Fig. 1　Division operation by levelled simultaneously.

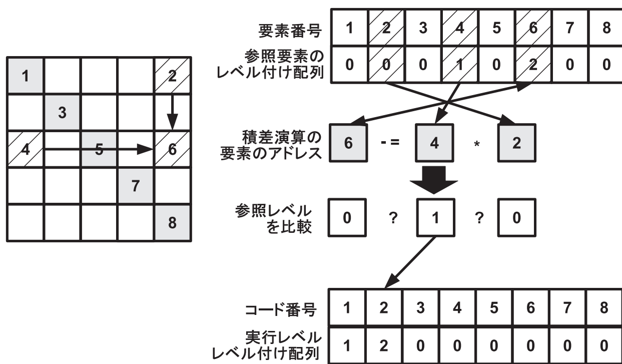

図 2　拡張ベクトル化 LU 分解法による積差演算のレベル付け
Fig. 2　Multiplication-substruct operation by levelled simultaneously.

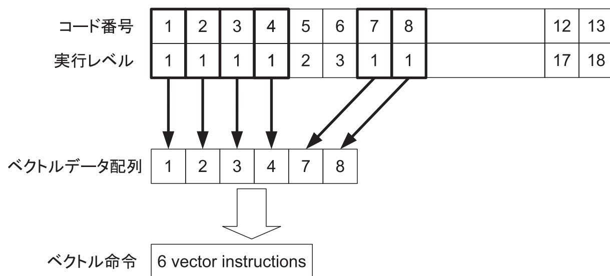

図 3　同時レベル付けによるベクトル化
Fig. 3　Vectorization by levelled simultaneously.

行列であるレベル格納行列（参照行列）を 0 に初期化する．

(2) LU 分解をシンボリック分解によってシミュレートする．シンボリック分解中の演算で用いる各要素の実行レベルを確認する．確認した実行レベルの中で，最大のレベルを持つ要素のレベルをその演算命令の実行レベルとして更新要素に +1 した値を設定する．

(3) 代入計算のレベル付けは，分解処理のレベル付けと同じくシンボリック分解する．このとき，代入計算の演算の追い越しを防ぐために，通常行単位で行うレベル付けを列単位で行い，計算する必要のある未知数の計算が終了したら，その演算要素のレベルを +1 する．

(4) すべての命令の実行レベルを設定したら，図 3 に示す実行レベルのレベル付け配列を用いて実行レベルの小さい命令から順に同一の実行レベルが設定されている命令を抽出する．対象とする実行レベルの命令が見つかれば，ベクトルデータ配列に命令を格納する．すべての同一の実行レベルを持つ命令を抽出したら，ベクトルデータ配列に格納した命令から同一レベル数分の並列度を持つベクトル命令を生成する．設定されたレベルが，同一レベルとして設定された演算は互いに依存関係がないため，並列に実行できる．

このようにベクトル命令の生成は，実行レベルの情報を用いてベクトル化する．このため，本手法で生成するベクトル命令は異なる演算を含む命令が抽出される可能性がある．

3. CUDA GPU のアーキテクチャを考慮した並列化手法

拡張ベクトル化 LU 分解法によって生成されるベクトル命令は，実行レベルが同一の異なる演算の命令を含む可能性がある．異なる演算の命令を含むベクトル命令を CUDA GPU で処理する場合，ベクトル命令は，CUDA GPU の処理の単位であるワープに分割され，ワープ単位で処理される．同一ワープで異なる命令を処理する場合，ワープダイバージェントという分岐処理が発生して，実行効率が低下する．CUDA GPU を用いてランダムスパース方程式求解を高速化するためには，多くの並列性を確保しつつ同一演算のみからなるベクトル命令を生成して，分岐命令を削減することが重要である．また，拡張ベクトル化 LU 分解法はマシンコードを生成する．しかし，マシンコードを CUDA GPU で直接実行することはできない．CUDA GPU で拡張ベクトル化 LU 分解法で抽出する命令の並列性を利用するためには，命令を抽出して配列に格納し，命令実行時に配列中の情報を解釈する必要がある．

CUDA で同一演算の命令を実行する拡張ベクトル化 LU 分解法を実装するために，提案手法では，命令を格納した配列から同時に実行する命令を抽出するときに，拡張ベクトル化 LU 分解法が命令を抽出する条件に同一演算命令であるという条件を付加する．**図 4** に，提案手法を用いてランダムスパース方程式を求解する手順を示す．図 4 中のシンボリック分解では，拡張ベクトル化 LU 分解法が抽出した演算の種類の情報と係数行列のインデックスの情報を**図 5** のように配列に格納する．本論文では，図 5 のように配列に格納した命令を，通常の命令と区別するために命令データと呼ぶ．格納された命令データを CUDA GPU で処理するために，CPU 上で，同時に実行可能でかつ同一演算の命令データの抽出を逐次的に行い，同時に実行可能な演算の命令データ数（ベクトル長）をカウントする．カウントしたデータは，演算の命令データを並列に実行するためのベクトル長の情報として付与することでベクトルデー

© 2014 Information Processing Society of Japan

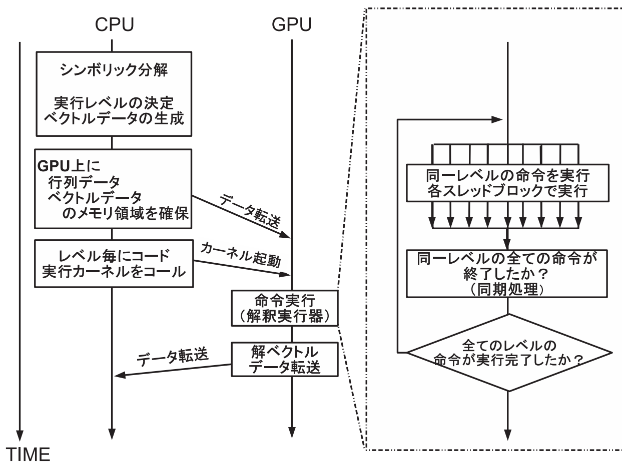

図 4　提案手法の実行手順
Fig. 4　Execution procedure of the proposed method.

除算演算命令	演算の種類	更新要素アドレス	参照要素アドレス	
積差演算命令	演算の種類	更新要素アドレス	参照要素アドレス	参照要素アドレス

図 5　命令データ
Fig. 5　Instruction data.

タを生成する．生成したベクトルデータは，CUDA GPU 上で解釈実行器により処理する．

本手法は，異なる演算命令を分けてベクトル命令を生成するためベクトル発行回数は増加するが，ワープダイバージェントの発生が減少するため，求解を高速化できると考えられる．以下では，ワープダイバージェントを回避する命令の抽出方法と，CUDA GPU でベクトルデータを実行するためのベクトル命令を実行するカーネルの実装について述べる．

3.1　ワープダイバージェントを防止する命令の抽出手法

提案手法では，同一演算のみで構成されるベクトルデータを生成するために，同一演算の命令データを抽出する．提案手法の命令抽出手法を**図 6** に示す．本手法は，まず，シンボリック分解により生成した演算の命令データに実行レベルを設定する．次に，実行レベルの低い命令から各命令の実行レベルと演算の情報が同じである命令データをベクトルデータ配列に割り当てる．もし，割り当てた演算の種類が異なる場合は，直後に実行するベクトルデータに割り当てる．以下に，同一演算の命令データの抽出方法とベクトルデータ生成の手順を示す．

(1) LU 分解をシミュレートするシンボリック分解を行い，各演算の命令データの実行レベルを決定する．

(2) シンボリック分解が終了したら，抽出した実行レベルの情報を用いて，図 6 に示すように実行レベルの低い順から実行可能な演算の命令データをベクトルデータ配列に格納する．このとき，同一演算の命令データの抽出を可能にするため，実行レベルごとに最初に割当てが行われた演算命令の情報を用いて命令データをベクトルデータ配列へ格納する．最初に割り当てた命令データと異なる演算命令の情報を持つ命令データは，その演算の実行レベルを +1 した値を設定して直後に実行するベクトルデータ配列へ格納する．

(3) ベクトルデータ配列へ格納された命令を CUDA GPU 上で実行可能なように演算の種類とベクトル長の情報を保持したベクトルデータを生成する．

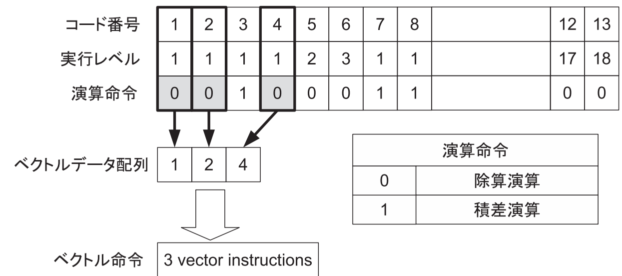

図 6　同時レベル付けによる同一演算のベクトル化
Fig. 6　Vectorization of identical operation by levelled simultaneously.

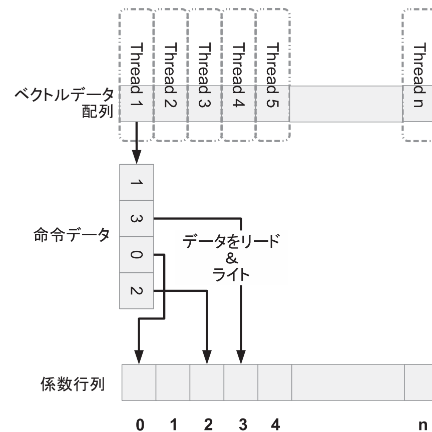

図 7　ベクトルデータの解釈実行器カーネル
Fig. 7　Interpretive-execution kernel of vector data.

このような手順でスケジューリングすることで，同一演算のみで構成されるベクトルデータと CUDA GPU 上で実行可能なベクトルデータを生成する．

3.2　ベクトル命令の実行カーネルの実装

本提案手法では，ベクトルデータ配列に格納された演算の命令データを解釈しながら求解に必要な係数行列のデータを用いて演算を処理することで方程式を求解する．解釈実行器の実行カーネルを**図 7** に示す．解釈実行器は，配列に格納されている演算の命令データをもとに求解に必要な演算の情報と要素の情報を配列から読み出し，演算を処理する．以下に，CUDA GPU 上で実装する解釈実行器の手順を示す．

(1) 解釈実行器は，CPU から割り当てられたベクトルデー

タを用いて演算の種類の情報から対応する演算のカーネルを起動する．カーネルは，ベクトルデータが保持している同時に実行可能な演算の命令データ数の情報を用いてベクトル長分のスレッドブロックを起動し，1 スレッドに 1 命令を割り当てる．

(2) 各スレッドは，演算の命令データから演算に必要なデータのアドレスを計算し求解処理に必要な係数行列のインデックスデータをロードする．

(3) ロードしたインデックスデータを用いて，解釈実行器は，係数行列のアドレスを計算し，アドレスを用いて演算に必要な要素データをロードする．係数行列のデータを間接アクセスによりグローバルメモリからレジスタへロードする．

(4) 解釈実行器は，演算の命令データに格納されている演算の情報を識別し，ロードしたデータを演算する．

(5) すべての実行レベルのベクトルデータの処理が終了するまで，手順 (2)〜手順 (4) を繰り返す．

4. 評価

CUDA GPU のワープダイバージェントを防ぐ拡張ベクトル化 LU 分解法の命令抽出手法の有効性を示すために，電子回路などの解析過程で生成されるランダムスパース方程式を求解し，実行時間を評価する．本評価の拡張ベクトル化 LU 分解法は，同一実行可能な命令データからベクトルデータを生成して，解釈実行器で処理する．

評価には，Florida Sparse Matrix Collection [13]，Matrix Market [14] の Matrix Market の行列生成スクリプトを用いて生成した問題，および，回路問題を用いる．**表 1** に評価環境，**表 2** に求解する問題を示す．本評価における数値精度は，本評価で用いる CUDA GPU では倍精度演算が無効化されているため，単精度演算を用いて評価する．表 2 中のグループと名前は，問題の種類であり，命令数は方程式の求解に必要な積差演算と除算演算の回数の和である．

4.1 拡張ベクトル化 LU 分解法と提案手法による評価

提案手法は，ワープダイバージェントの発生を防止するために同一の演算の命令データのみをベクトル化したベクトルデータを生成するので，求解に必要なベクトルデータ数やワープの実行効率に影響する．そこで提案手法の有効性を評価するために，拡張ベクトル化 LU 分解法と提案手法のベクトル長と分岐回数，方程式求解時間を測定する．

まず，ベクトルデータの命令データ数であるベクトル長を**表 3** に示す．表 3 中の MAX LEVEL は，求解に必要なベクトルデータの発行回数，MAX VECTOR は求解に必要なベクトルデータの中で最大のベクトル長である．表 3 より，提案手法は，拡張ベクトル化 LU 分解法に対し，求解に必要なベクトルデータの発行回数は平均約 1.12 倍増加するが，最大のベクトル長は同一であるという結果になった．以上より，提案手法は，拡張ベクトル化 LU 分解法と同様に多くのベクトル化を引き出せることが確認できた．

次に，ワープダイバージェントを防止したことによる分岐回数と実行時間を評価した結果を**表 4** に示す．表 4 中の分岐回数は，CUDA Tool Kit の NVIDA Visual Profiler（nvvp）を用いて図 4 に示すようにレベルごとに命令実行カーネルをコールして実行したときの回数である．表 4 中の分岐回数の平均値は式 (2)，最大値は式 (3) により求める．

$$\text{分岐回数の平均} = \frac{\sum_i \text{ベクトルデータ } i \text{ の分岐回数}}{\text{ベクトルデータの発行回数}} \tag{2}$$

$$\text{最大分岐回数} = \max_i \text{ベクトルデータ } i \text{ の分岐回数} \tag{3}$$

表 4 より，提案手法は，拡張ベクトル化 LU 分解法に対して，すべての問題で分岐回数の削減が確認できた．分岐回数を削減することで提案手法は，ワープダイバージェントの発生回数が減少し，多くの問題で求解時間が高速化したと考えられる．特に circuit_4 は，分岐回数を削減することで最大約 1.6 倍の実行時間の高速化が得られた．

次に，ワープダイバージェントを防止し分岐処理を削減することで得られたワープの実行効率を評価する．**表 5** に nvvp で計測したワープの実行効率を示す．表 5 中の min はベクトルデータの処理時に最も悪い効率が得られたときの実行効率，max はベクトルデータの処理時に最も良い効率が得られたときの実行効率，avg はすべてのベクトルデータを処理して得られた実行効率の平均である．評価の結果，拡張ベクトル化 LU 分解法に対して提案手法は，平均で約 1.5 倍，実行効率が最も悪い場合でも約 2.1 倍の効率化が得られたが，memplus は高速化率が減少した．CUDA

表 1　評価環境
Table 1　Evaluation environment.

CPU	Intel Core i7 3930K
Memory	64 GB
GPU	Geforce GTX 580 3 GB
CUDA Drive / Runtime Version	5.5 / 5.0
GPU L1 Chache Size	48 KB
OS	CentOS 6.3
Kernel	2.6.32
GCC Version	4.4.7

表 2　評価問題
Table 2　Evaluation data.

Group	Name	行列サイズ	非零要素数	命令数
Random	Random	8,000	16,000	23,642
Hamm	add32	4,960	19,848	58,068
Hamm	memplus	17,758	99,147	1,238,173
Bomhof	circuit_3	12,127	48,137	2,307,224
Bomhof	circuit_4	80,209	307,604	7,277,215

© 2014 Information Processing Society of Japan

表 3　拡張ベクトル化 LU 分解法と提案手法のベクトル長
Table 3　Vector length of expected LU factorization method and the proposed method.

Group	Name	拡張ベクトル化		提案手法	
		MAX LEVEL	MAX VECTOR	MAX LEVEL	MAX VECTOR
Random	Random	94	8,371	109	8,371
Hamm	add32	128	9,756	147	9,756
Hamm	memplus	884	38,173	1,063	38,173
Bomhof	circuit_3	3,213	20,298	3,499	20,298
Bomhof	circuit_4	14,343	110,626	14,581	110,626

表 4　拡張ベクトル化 LU 分解法と提案手法の分岐回数と実行時間
Table 4　Execution time and the number of branchings expected LU factorization by the proposed method.

Name	拡張ベクトル化 LU 分解法 実行時間 [ms]	拡張ベクトル化 LU 分解法 分岐回数 平均：最大	提案手法 実行時間 [ms]	提案手法 分岐回数 平均：最大	高速化率 [倍]
Random	0.21	68：1,582	0.20	36：534	1.05
add32	0.32	105：1,989	0.31	43：640	1.11
memplus	1.90	240：5,972	1.96	92：1,056	0.97
circuit_3	9.10	136：3,618	6.09	67：1,056	1.49
circuit_4	45.07	102：17,416	28.21	66：15,779	1.60

表 5　Warp の実行効率
Table 5　Execution efficiency of warp.

Name	拡張ベクトル化 [%]			提案手法 [%]		
	min	avg	max	min	avg	max
Random	51.8	68.9	99.9	86.2	97.2	100
add32	40.7	83.2	100	86.2	96.9	100
memplus	52.7	84.7	100	86.2	97.9	100
circuit_3	47.6	71.1	100	86.2	98.9	100
circuit_4	58.8	69.1	100	88.6	99.9	100

表 6　命令の割合
Table 6　Ratio of instructions.

Name	除算演算 [%]	積差演算 [%]
memplus	5.7	94.3
circuit_4	3.6	96.4

GPU は，ワープ単位で処理が行われるため，ワープの実行効率を高める並列性を抽出することで処理を高速化できる．本手法は，演算命令ごとにベクトルデータを生成するため，ワープの処理単位以下の並列度を持つベクトルデータが生成される可能性がある．このため，高速化率が最も高い問題である circuit_4 と最も低い問題である memplus の問題におけるワープの処理単位である 32 並列未満のベクトル長のベクトルデータの割合を評価する．評価の結果，circuit_4 で約 0.5%，memplus では約 19%であった．これは，並列度が低いベクトル長の割合が多いほど，CUDA GPU を効率的に動作させるための並列度が得られず高速化できないと考えられる．

本提案手法は，演算命令の種類ごとにベクトル化するため，演算命令の割合がベクトルデータの並列度に影響すると考えられる．そこで，memplus と circuit_4 における 2 種類の演算の割合を**表 6** に示す．表 6 より，memplus における除算演算が全体の約 5.7%であったのに対し，積差演算が全体の約 94.3%であった．また，最も高い高速化率が得られた circuit_4 では，除算演算が全体の約 3.6%であったのに対し，積差演算が全体の約 96.4%であった．以上のことから，全体の計算に占める除算の割合が低い問題では，提案手法によって異なる演算を分割すると，低い並列度を持つベクトルデータが多く生成される．このような問題を提案手法により求解すると実行レベルが増加し演算速度が低下することから，積差演算の割合に対して除算演算の割合が極端に低い問題では，拡張ベクトル化 LU 分解法の命令抽出方法を適用することも考慮する必要がある．

また，評価結果より，提案手法は分岐を防止する手法であっても分岐が発生していることが分かる．これは，スレッド情報のデータである threadIdx による分岐処理が考えられる．

4.2 SuperLU と提案手法による実行時間の評価

提案手法を用いたランダムスパース方程式求解の有効性を評価するため，ランダムスパース LU 分解法において，高速に求解可能なオープンソースのソルバの 1 つである SuperLU_MT [15] を表 1 の環境を用いて 6 並列で実行したときの求解時間と，提案手法を実行したきの求解時間を評価する．提案手法の求解時間は，図 4 に示すシンボリック分解とベクトルデータ生成に要したスケジューリング時間，求解に必要なベクトルデータと係数行列のデータを GPU

© 2014 Information Processing Society of Japan

表 7　提案手法と SuperLU_MT の LU 分解法の実行時間
Table 7　Execution time of SuperLU_MT and the proposed method.

Group	Name	SuperLU_MT 実行時間 [ms]	提案手法 実行時間 [ms]	提案手法 スケジューリング時間 [ms]	高速化率 [倍]
Random	Random	8.62	0.20	11.61	43.1
Hamm	add32	3.73	0.31	7.84	12.03
Hamm	memplus	299.54	1.96	420.32	152.82
Bomhof	circuit_3	272.40	6.09	1,434.86	44.73
Bomhof	circuit_4	–	28.21	13,196.58	–

表 8　CULA ルーチンと提案手法による実行時間
Table 8　Execute time by CULA routine.

Name	CULA [ms]	提案手法 [ms]
Random	485.33	0.20
add32	188.22	0.31
memplus	2,997.07	1.96
circuit_3	1,449.53	6.09
circuit_4	–	28.21

上に転送し，ベクトルデータを CUDA GPU で処理する時間を測定する．SuperLU_MT と提案手法における実行時間を**表 7** に示す．表 7 中の横線は，メモリエラーのため測定できなかったことを表す．評価の結果，SuperLU_MT を用いて方程式を求解した場合に対して，提案手法はすべての問題において処理時間の高速化が確認できた．特に memplus は，最大約 152 倍の高速化が確認できた．また，SuperLU_MT での実行時間に対するベクトルデータ生成にかかった時間は，最大 circuit_3 で約 5 倍であった．本論文で対象とする回路問題の過渡解析は，非線形連立微分方程式を解くために何度も同一の行列構造を持つランダムスパース方程式を解く．よって，提案手法は，初回のみ求解に必要なベクトルデータを生成して，生成したベクトルデータを使い方程式求解を繰り返すことで，ベクトルデータ生成にかかる割合を十分少なくすることができ，非線形連立微分方程式求解を高速化できると考えられる．

4.3　CULA と提案手法による実行時間の評価

提案手法の有効性を客観的に評価するために，CUDA GPU 向けの方程式求解ソルバを用いて求解時間と，提案手法の求解時間を評価する．CUDA GPU の直接法によるランダムスパース疎行列向け方程式求解ソルバは公開されていないため，密行列向けの求解ソルバによって，行列構造をブロック化して多くの並列性を引き出し方程式求解を高速化する商用ライブラリの 1 つである CULA [16] を用いる．提案手法と CULA ソルバの実行時間を**表 8** に示す．表 8 中の横線は，メモリエラーのため測定できなかったことを表す．評価の結果，すべての問題において提案手法は CULA よりも高速に求解できることが確認できた．特に circuit_3 において，提案手法は CULA に対して約 238 倍の高速化が得られた．circuit_3 で高い高速化が得られたのは，係数行列のスパース性が非常に高いからである．CULA は，ブロック化した際に非零要素が存在しない，または少ないブロックも並列化して計算するため，零要素に対する無駄な演算が多く実行される．提案手法は，演算の命令データを用いることで求解に必要な演算のみを実行するため，無駄な演算を削減し高い高速化が得られたと考えられる．

5.　おわりに

本論文では，CUDA GPU を用いて実非対称なランダムスパース方程式求解を高速化するために，同一の演算命令を抽出することでワープダイバージェントを防止する手法を提案し，評価した．評価の結果，提案手法は拡張ベクトル化 LU 分解法に対して，最大約 1.6 倍高速に求解できることが確認できた．今後の課題として，提案手法をさらに高速化するために，並列化によるスケジューリング時間の短縮，および，本提案手法を電子回路シミュレーション SPICE に組み込むことで過渡応答計算を高速化できると考えられる．

参考文献

[1] NVIDIA Corporation: NVIDIA CUDA C Programming Guide v5.0, available from ⟨https://developer.nvidia.com/category/zone/cuda-zone⟩.

[2] Krawezik, G.P. and Poole, G.: Accelerating the ANSYS Direct Sparse Solver with GPUs, *2010 Symposium on Application Accelerators in High Performance Computing* (*SAAHPC '10*) (2010).

[3] Demir, V.: Graphics Processor Unit (gpu) Acceleration of Finite-difference Frequency-domain (fdfd) Method, *Progress In Electromagnetics Research M*, Vol.23, pp.29–51 (2012).

[4] 鹿毛哲郎，下郡慎太郎：ベクトル計算機による高速回路解析のためのベクトル化処理技法，電子情報通信学会論文誌，Vol.70, No.8, pp.1597–1606 (1987).

[5] Quarles, T.L.: *The SPICE3 Implementation Guide*, Electronics Research Laboratory, College of Engineering, University of California (1989).

[6] 三浦道子，名野隆夫，盛　健次：回路シミュレーション技術と MOSFET モデリング，リアライズ理工センター (2003).

[7] 前川仁孝，高井峰生，伊藤泰樹，西川　健，笠原博徳：スタティックスケジューリングを用いた電子回路シミュ

© 2014 Information Processing Society of Japan

レーションの粗粒度/近細粒度階層型並列処理手法，情報処理学会論文誌，Vol.37, No.10, pp.1859–1868 (1996).

[8] Christen, M., Schenk, O. and Burkhart, H.: General-Purpose Sparse Matrix Building Blocks Using the NVIDIA CUDA Technology Platform, *Proc. 1st Workshop on General Purpose Processing on Graphics Processing Units*, Northeastern University, Boston (2007).

[9] Jalili-Marandi, V., Zhou, Z. and Dinavahi, V.: Large-Scale Transient Stability Simulation of Electrical Power Systems on Parallel GPUs, *2012 IEEE Power and Energy Society General Meeting*, pp.1–11 (2012).

[10] Chen, X., Wu, W., Wang, Y., Yu, H. and Yang, H.: An EScheduler-Based Data Dependence Analysis and Task Scheduling for Parallel Circuit Simulation, *IEEE Trans. Circuits and Systems II: Express Briefs*, Vol.58, No.10, pp.702–706 (2011).

[11] Chen, X., Wang, Y. and Yang, H.: Parallel Circuit Simulation on Multi/Many-core Systems, *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum* (*IPDPSW*), pp.2530–2533 (2012).

[12] Yamamoto, F. and Takahashi, S.: Vectorized LU Decomposition Algorithms for Large-Scale Circuit Simulation, *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, Vol.4, No.3, pp.232–239 (1985).

[13] Davis, T.A. and Hu, Y.: The University of Florida Sparse Matrix Collection, *ACM Trans. Math. Softw.*, Vol.38, No.1, pp.1–25 (2011).

[14] Matrix Market, available from ⟨http://math.nist.gov/MatrixMarket/⟩.

[15] Demmel, J.W., Eisenstat, S.C., Gilbert, J.R., Li, X.S. and Liu, J.W.H.: A Supernodal Approach to Sparse Partial Pivoting, *SIAM J. Matrix Analysis and Applications*, Vol.20, No.3, pp.720–755 (1999).

[16] CULA Tools: GPU Accelerated LAPACK, available from ⟨http://www.culatools.com/⟩.

富永 浩文　（学生会員）

1984 年生．2007 年千葉工業大学情報科学部情報工学科卒業．2009 年同大学大学院情報科学研究科情報科学専攻博士前期課程修了．2010 年同大学院情報科学研究科情報科学専攻博士後期課程入学．主として，GPU 等のアクセラレータを用いた各種アプリケーション高速化の研究に従事．

前川 仁孝　（正会員）

1967 年生．1990 年早稲田大学理工学部電気工学科卒業．1992 年同大学大学院理工学研究科電気工学専攻修士課程修了．1993 年日本学術振興会特別研究員．1994 年早稲田大学理工学部助手．1998 年千葉工業大学情報工学科講師．2002 年同大学助教授．2011 年同大学教授，現在に至る．博士（工学）．主として，各種アプリケーションの並列処理の研究に従事．

© 2014 Information Processing Society of Japan