千葉工業大学

博士学位論文

GPUのアーキテクチャを考慮した
数値解析の高速化に関する研究

平成31年3月

富永 浩文

要旨

本論文は，GPU を用いた数値解析において GPU の実行効率を高めることを目的とする．防災や天気予報，製品開発などの幅広い分野で用いられる数値解析を始めとした科学技術計算は，大規模かつ精度の高いシミュレーションが求められている．高速かつ精度の高い計算できる計算機アーキテクチャとして，GPU が注目されている．GPU は，演算コアを多数搭載し，広帯域なメモリバンド幅のメモリを持つ SIMT アーキテクチャである．SIMT アーキテクチャは，複数のデータを同一の命令で処理することで高い実行性能を得られる．このため，GPU を用いる数値解析には，GPU のアーキテクチャに適した計算アルゴリズムが必要となる．

そこで，本論文では，GPU を用いた数値解析を高速化するために，アーキテクチャの特性を活かして GPU の実行効率を向上する手法を提案し，その有効性を評価する．以下に本論文の各章の概要を述べる．本論文は全 6 章より構成される．まず，第 1 章「序論」では，本研究における背景および従来研究について述べ，提案手法の目的や位置づけを明らかにする．

第 2 章「CUDA」では，GPU のハードウェア構成と CUDA プログラミングについて述べる．CUDA のプログラミングモデルは，スレッド階層とメモリ階層から構成される．このため，GPU の実行効率を高めるためには，CUDA の階層的構造に合わせてプログラムの局所性を抽出する必要がある．

第 3 章「メモリアクセスの局所性の向上」では，レジスタやシェアードメモリなどの階層的なメモリを用いる最適化手法について提案する．GPU は，多くのスレッドが同時に

起動できるように多くのレジスタを搭載する．このため，大量のスレッドを同時に実行でき，メモリアクセスのオーバーヘッドを隠蔽することができる．メモリアクセスのオーバーヘッドの削減には，シェアードメモリを利用する手法があり，高い効果が得られている．メモリアクセスのオーバーヘッドをより削減するためには，GPU のもつ多くのレジスタを効率的に利用することで計算をさらに高速化できる．本章では，格子ボルツマン法を題材として，局所性の高いデータをレジスタに保持することで高速化する手法を提案する．本提案手法により，テンポラルブロッキングを用いた手法と用いない手法で，最大約 7.36 倍の高速化が確認できた．

第 4 章「並列性の抽出による高速化」では，GPU の実行単位であるワープに最適化する形で並列性を抽出する手法を提案する．命令の実行単位は，ワープと呼ばれる 32 スレッドのまとまりで命令を実行する．このため，異なる演算が同一のベクトル命令に抽出されると，実行効率が低下する．そこで，本章では，並列性の低い問題であるランダムスパース方程式求解の LU 分解法を題材とし，命令の並べ替えを行うことでワープの実行効率を最大化する手法を提案する．本章の提案手法により，CUDA 向けの数値計算ライブラリである CULA の LU 分解ルーチンを利用した手法にくらべ提案手法は，最大約 238 倍の高速化が確認できた．

第 5 章「並列度に応じたハイブリッド並列化手法による高速化」では，CPU と GPU で物理メモリを共有するアーキテクチャを利用し，並列度に応じて CPU と GPU を使い分けることで計算を高速化する手法を提案する．ハイブリッド並列化による計算は，CPU と GPU のそれぞれのメモリに局所性のあるデータを割当てる．一方，局所性のないデータを割当てて計算する場合は，バスを介したデータ転送が必要であり，性能が低下する．バスを用いないアーキテクチャとして，単一の物理メモリを共有して利用できるヘテロジニアスマルチコアアーキテクチャがある．本アーキテクチャは，バスを介さずにデータのやり取りができる．本章の提案手法では，拡張ベクトル化 LU 分解法を題材として，ベ

クトル長に応じてCPUとGPUを切り替えることで，計算を高速化する手法を提案する．本提案手法により，全ての実行レベルをGPUで行う手法にくらべ，提案手法は，最大約26倍の高速化が確認できた．

最後に第6章では，提案手法と評価結果をまとめ論文全体を総括する．

Summary

The aim of this paper is the method improving the execution efficiency using the characteristic of the architecture of GPU to speed up a numerical simulation. A numerical simulation of the various fields such as accident prevention, weather forecast, and product development needs a large-scale and highly accurate simulation. GPU is the processor which can do highly precise numerical value calculation at high speed. This processor is the SIMT architecture with a lot of calculation cores and a high memory bandwidth. SIMT architecture can get the high execution performance by treating more than one data with an identical instruction. Therefore, speeding of numerical calculation using GPU is the necessity to optimize calculation algorithm to the architecture of GPU.

The proposed method improves the execution efficiency using the characteristic of the architecture of GPU to speed up a numerical analysis. This paper is composed of six sections as follows.

In first, section 1 refers to the summary of this study and the necessity of optimizing of GPU architecture.

Section 2 describes the hardware constitutions of GPU and the CUDA programming. The CUDA programming model consists of the thread hierarchy and the memory hierarchy. It is necessary to extract the locality of the programs based on the hierarchical structures of CUDA in order to improve the execution efficiency of GPU.

Section 3 describes optimization method using hierarchical memory of a register and a

shared memory. GPU can launch many threads by many registers, and can conceal an overhead of a memory access. There is the method using a shared memory to reduce an overhead of memory access, and this method obtains the highly execution efficiency. In addition, it is possible to use many registers of GPU efficiently and reduce an overhead of memory access more efficiently. As a result of the evaluation, the speedup ratio of the proposed method compared with not considering the locality using registers is about 7.36 times faster on the maximum.

Section 4 describes a method to extract parallelism in a form optimized for the warp of execution unit of GPU. GPU executes an instruction by the SIMT form by the unit of 1 warp/32 threads. Therefore, when an order different in execution of the vector instruction is included, a performance of execution efficiency of gpu falls. This proposed method optimizes and applies EMVA of LU decomposition in SIMT of GPU. Optimization technique increases the parallelism of the instruction by EMVA and generates a vector instruction only of an identical instruction. As a result of the evaluation, the speedup ratio of the proposed method compared with the CULA LU decomposition routine is about 238 times faster on the maximum.

Section 5 describes a method to speed up the calculation by using the CPU and GPU properly according to the degree of parallelism. A numerical calculation by hybrid parallelism assigns data of a locality to a memory of CPU and GPU. In the case of no locality in data, the performance falls by communication using a bus. Therefore, the data communication by a bus is a high overhead. There is a heterogeneous multi-core processor in the architecture in hybrid parallel processing having no bus of between CPU and GPU. This architecture shares one physical memory between CPU/GPU, and the data can be forwarded to high speed. The method to propose speed up EMVA calculation using het-

erogeneous multi-core processor. EMVA is the method by which the vector length changes big during calculation. As a result of the evaluation, the speedup ratio of the proposed method compared with the method of doing all execution levels with the GPU is about 26 times faster on the maximum.

Finally, section 6 describes the conclusions of this paper.

目 次

図 目 次

図一覧

表 目 次

第1章

序論

数値解析は，物理現象や金融や経済の変動のように微分方程式で表される連続的な変化を数値的に解析する手法である．数値解析では，現象を数理モデルにモデル化し，数学的に離散化することでコンピュータを用いた解析を可能にする．近年では，気象などの環境シミュレーション [1][2][3][4][5]，だけでなく，防災や製品開発などの幅広い分野で用いられている [6][7][8][9][10][11]．数値解析を用いたシミュレーションは，大規模かつ高精度であることが求められることが多い．例えば，災害シミュレーションでは，特定の地域だけなく広範囲の地域の複合的な被災状況を確認するために，解析点数の多い問題を解く必要がある．また，シミュレーションの精度を向上するためには，時間や空間の離散化幅を小さくする必要があり，演算回数の増加が問題となる．演算回数の増加による問題の大規模化は，計算時間の増加に繋がるため高速化が求められている．大規模でかつ精密なシミュレーションを高速に実現するために，並列処理による数値解析の高速化の研究が多く行われている．数値計算の高い並列性を利用できるアーキテクチャに GPU が注目されている [12][13].

GPU は，Single Instruction Multiple Thread（SIMT）型の超並列なアーキテクチャである [14]．本アーキテクチャは，多くの軽量なインオーダーコアを持ち，これらを並列に動作させるために，多くのレジスタを持つ．GPU は，多くのスレッドを動作させる事が可能なアーキテクチャであるため，専用の開発環境プラットフォームが複数提供されている [15][16][17]．中でも細かいチューニングなどが可能である高いプログラミング性の自由

度と高度に最適化されたコンパイラにより高い高速化が得られる開発環境プラットフォームの一つに NVIDIA より開発，提供されている Compute Unified Device Architecture（CUDA）と呼ばれる並列コンピューティングプラットフォーム・プログラミングモデルがある．CUDA による GPU プログラミングアーキテクチャは，スレッド階層とメモリ階層から構成される．このため，CUDA による数値計算は，GPU のアーキテクチャの特性を利用することでより計算を高速化することができる [2][3][18] [19][20].

そこで，本論文では，アーキテクチャの特性に合わせた数値解析を高速化するために，まずメモリ階層の効率的な利用，次に SIMT 実行形式に対応する並列性の抽出方法，最後に並列度に応じた計算の CPU/GPU ハイブリッド切替手法について，それぞれの最適化手法を提案する．

第2章

CUDA

2.1 はじめに

本章では，本研究の要になるCUDAアーキテクチャ，及びCUDAプログラミングモデルについて解説する．

Compute Unified Device Architecture（CUDA）は，NVIDIA社が提供するGraphics Processor Unit（GPU）を用いたコンピューティングプラットフォームであり，C言語を拡張したプログラミングモデルである．NVIDIA製のTeslaやQuadro，Geforceなどの GPU，Jetsonなどの GPUを含むSoCとCUDAを利用できる．CUDAは，これらの様々なアーキテクチャの違いを隠蔽しアプリケーションをハードウェアごとに最適化して実行することが可能である．GPUは，Single Instruction Multiple Threads（SIMT）型の超並列計算機アーキテクチャである．CUDAを用いた数値計算において最適な動作を実現するためには，アーキテクチャとソフトウェアの両面の特徴を理解することはとても重要となる．以下，2.2節ではCUDAのプログラミングモデル，2.2.1節，2.2.2節，2.2.3節ではCUDAプログラミングの最適化における重要な事項について述べる．

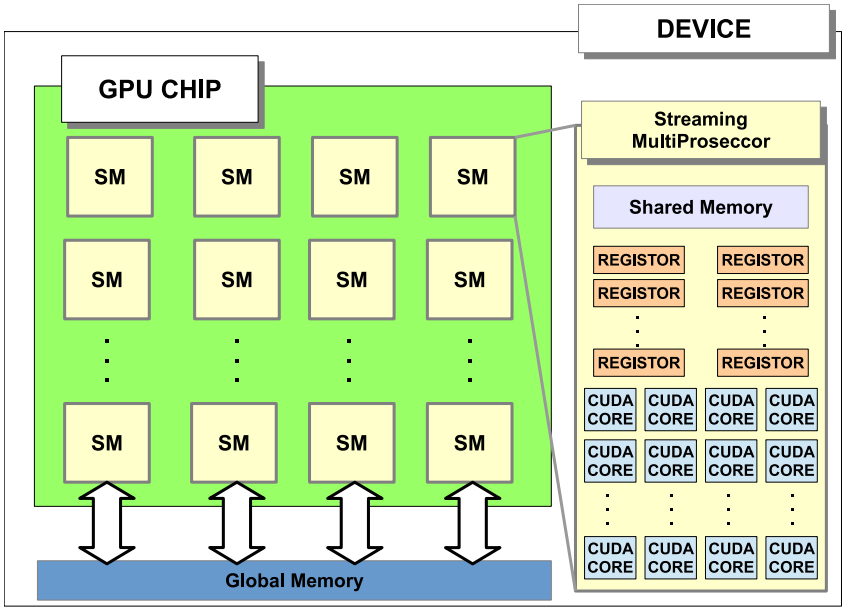

図 2–1 ： GPU アーキテクチャ

2.2　CUDA プログラミングモデル

CUDA 環境は，GPU と呼ばれる NVIDIA 社製のプロセッサを用いることで構築できる．GPU は，多くのインオーダー型の軽量の計算コアとこれらのコアを十分に動作させるための広帯域なメモリを搭載する．このため，数千から数万のスレッドを起動し，これらの大量のスレッドを同時に処理するために広帯域なメモリを使い多くのデータを効率良くアクセスすることで効率の高い実行が可能となる．図 2–1 に GPU（デバイス）アーキテクチャを示す．GPU アーキテクチャは，複数の計算コアである CUDA コア，レジスタとシェアードメモリで構成される Streaming Multiprocessor（SM）を複数搭載した GPU チップとグローバルメモリで構成される．レジスタのデータは，レジスタを占有した CUDA コアのみがアクセスでき，シェアードメモリのデータは SM 内の全ての CUDA コアがアクセスでき，グローバルメモリのデータは全ての SM がアクセスできる．これらのメモリは，アクセスできる範囲に制限があるだけでなく，容量や速度にも差がある．

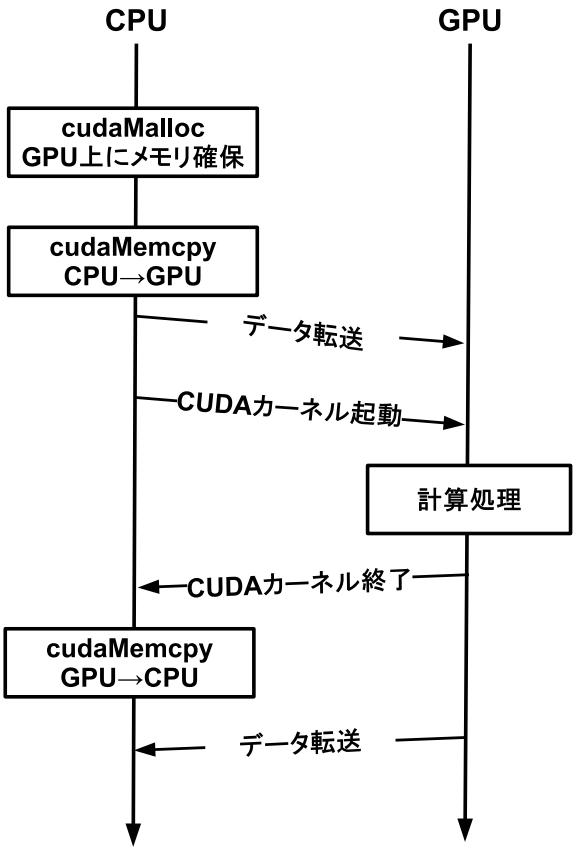

図 2–2 ： CUDA におけるカーネル起動とデータ転送

CUDA では，カーネルと呼ばれるプログラムを GPU 上で実行するために，実行する計算カーネルは，一般的に CPU がホストとなり GPU 上にカーネルを生成する．カーネルは GPU 上で実行するプログラムであり，必ずホストである CPU から起動する．ただし，CPU と GPU は，NVLINK ないし PCI-Express などのバスによって接続される．CPU と GPU は，それぞれ専用のメモリを持ち，物理的に異なるアドレスを持つ．このため，CUDA を用いるアプリケーションでは，CPU と GPU 間でメモリ転送が必須となる．図 2–2 に CUDA におけるカーネル起動とデータ転送の例を示す．図 2–2 のように，CPU 上のホストプログラムが cudaMalloc 関数を用いて GPU のグローバルメモリにデータ格納領域を確保する．cudaMemcpy 関数は，データを CPU から GPU へ転送するためにホス

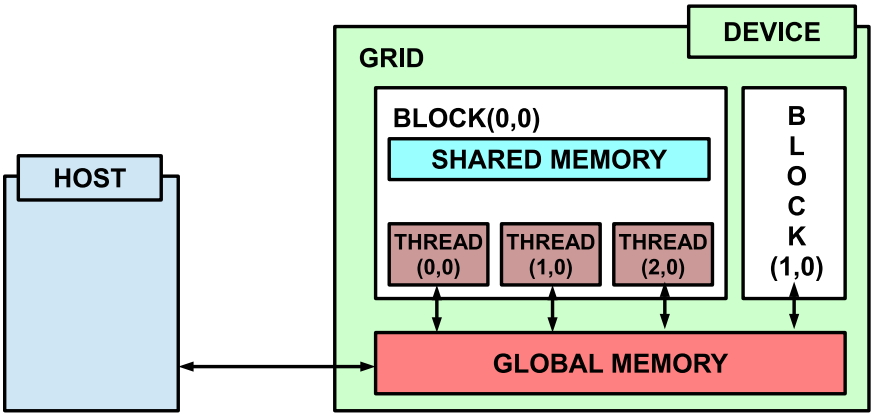

図 2–3 ： CUDA 環境

トプログラムが発行する命令である．転送するデータのホストメモリのアドレスとグローバルメモリのアドレス指定し，指定したバイト数の必要なデータをコピーする．データのコピーが完了するとホストプログラムは，カーネルを GPU 上に起動する．計算が終了すると GPU は，計算が完了したことを CPU に通知する．GPU の計算処理終了後に，ホストプログラムが cudaMemcpy 関数を実行するのは，GPU 上の計算結果をホストメモリにコピーし，CPU で計算結果を参照できるようにするためである．CUDA プログラミングでは，このように CPU と GPU で異なるアドレス空間を持つためデータ転送が必要になるため，CPU と GPU 間の通信と計算のオーバーラップや通信回数の削減が重要となる．

GPU は，数千から数万というスレッドを起動して超並列でデータを処理する．CUDA では，大量のスレッドを効率的に管理するために，GRID（グリッド），BLOCK（ブロック），THREAD（スレッド）から構成される階層的な構造でカーネルを起動し多くのスレッドを管理する．スレッドは処理の最小単位であり，スレッドブロックは複数のスレッドを束ねた単位であり，グリッドは複数のスレッドブロックを束ねた単位である．スレッド階層を，図 2–3 に示す．グリッド，ブロック，スレッドは，GPU アーキテクチャのデバイス，SM，CUDA コアにそれぞれ対応する．CUDA は，同時に複数のスレッドブロッ

クを同時に起動し実行する．多くのスレッドブロックを起動し実行することで，低速なグローバルメモリからのデータアクセスのオーバーヘッドを隠蔽し処理を高速化する．更に，図 2–1 で示したように SM にはスレッド間でデータを共有可能なシェアードメモリがある．スレッドブロック内のスレッド間で共有する必要のあるデータをシェアードメモリに配置することでグローバルメモリへのアクセスを軽減し，処理を高速化できる．このように，CUDA は階層的なメモリを用いることでメモリアクセスのオーバーヘッドを軽減するため，メモリアクセスの局所性が重要となる．また，CUDA はブロック単位で処理を行うが，スレッドブロック内で更に 32 スレッドごとにまとまって処理を実行する．このまとまりを Warp（ワープ）という．ワープごとに，命令の実行やメモリアクセスが行われることから，ワープの処理を意識する必要がある．

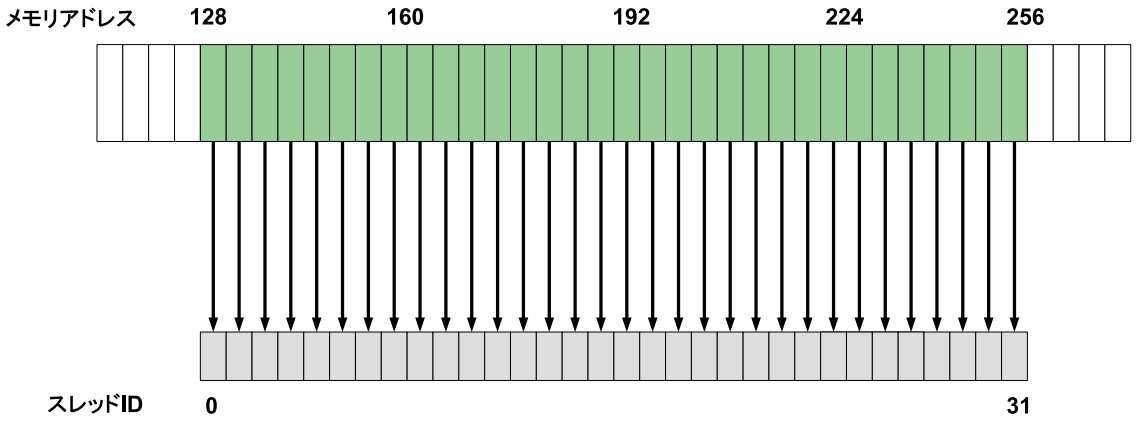

図 2–4 ： アラインアクセスとコアレスアクセス

2.2.1 アラインアクセスとコアレスアクセス

グローバルメモリは，ホストメモリからデータをロードした場合にまずはじめに格納されるメモリ領域である．このため，GPU 内で多くのメモリ容量を持つ．このメモリ容量は，2018 年現在の最新の GPU で約 32GB である．グローバルメモリは，多くのスレッドを並列に実行可能にするために広帯域であるが，アクセスコストに数百クロックを要する．このため，グローバルメモリへのアクセスは，少ないアクセスで効率的にデータアクセスを行う必要がある．メモリアクセスの単位は，32 バイトや 128 バイト単位でアクセスする．このため，この単位内にスレッドがアクセスするデータが含まれていることが最も効率良くメモリアクセスできるアラインアクセスとなる．また，ワープ内のスレッドが連続してアクセスするとコアレスアクセスとなり，効率良くアクセスできる．図 2–4 に，アラインアクセスとコアレスアクセス例を示す． 図に示すように本例のアクセスは，CUDA のスレッドの実行単位はワープ単位であるため，ワープ内の 32 スレッドが連続した領域をそれぞれロードすることでコアレスアクセスとなる．また，全てのスレッドが 128 バイトの範囲内のデータにアクセスしていることから一度のアクセスでメモリアクセスが完了できる．

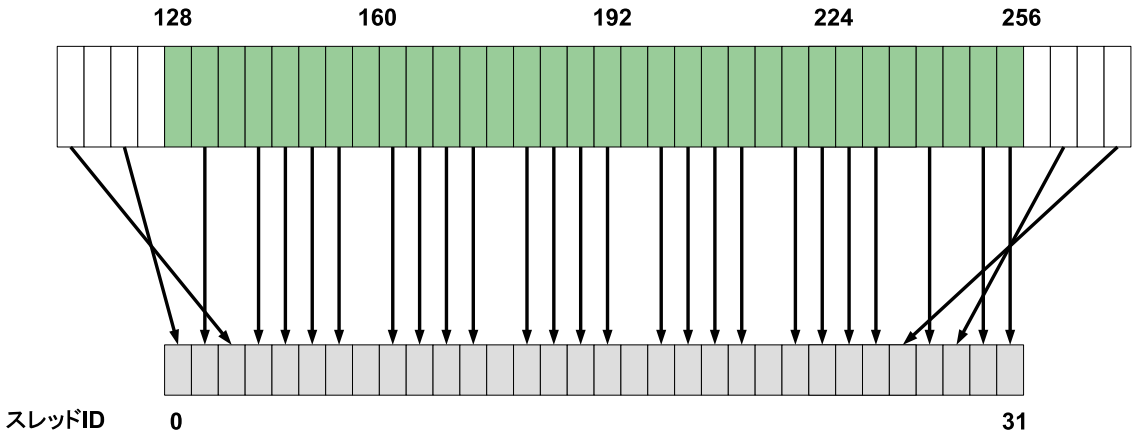

図 2–5 ： ミスアラインアクセスとアンコアレスアクセス

一方，ワープ内のスレッドが順にアクセスしないような場合はコアレスアクセスとならず，メモリアクセスの先頭アドレスから個別にアクセスする．このような場合，メモリトランザクションが複数回必要となる．さらに，メモリアクセス単位の 32 や 128 バイトのアクセス範囲を超えるような場合はアラインアクセスにならず，各メモリアクセスの先頭となるアドレスから個別にアクセスする．このような場合も，アンコアレスアクセスと同様に複数回のメモリトランザクションが必要となる．図 2–5 に，ミスアラインアクセスとアンコアレスアクセス例を示す． 図に示すように本例のアクセスは，ワープ内の 32 スレッドが連続しない領域を読み込むため，複数回のアクセスが行われ効率の悪いアクセスとなる．

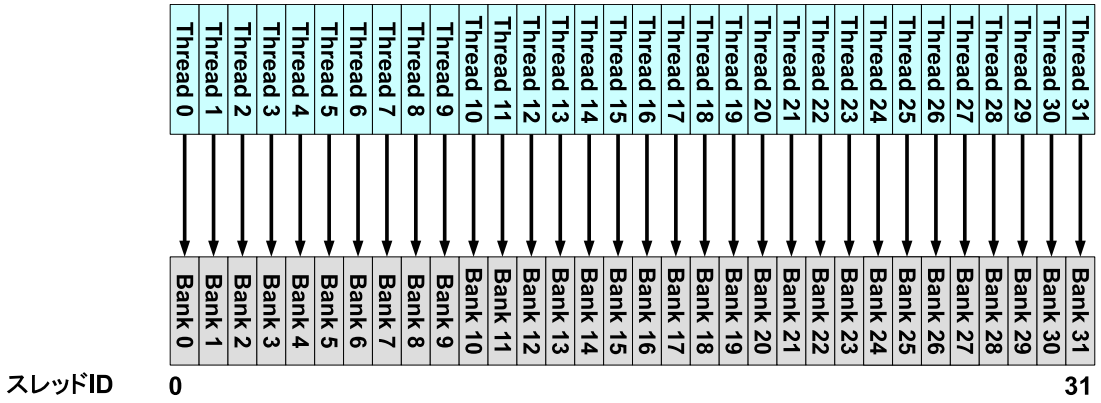

図 2–6 ： コンフリクトが起こらない並列アクセスパターン

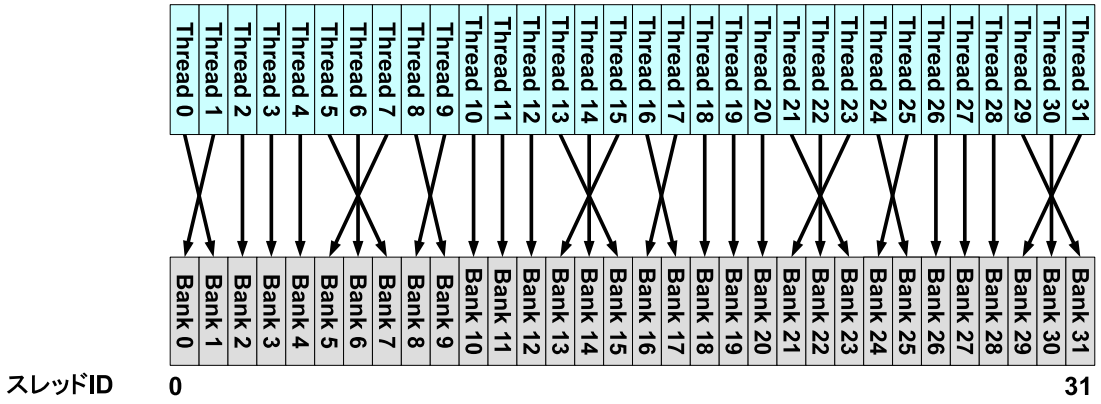

図 2–7 ： コンクリフトが起こらないランダムな並列アクセスパターン

2.2.2 バンクコンクリフト

シェアードメモリは，各 SMX 内にそれぞれ実装されておりスレッドブロック内の各スレッドが共有して利用できる．グローバルメモリよりも低遅延で利用できる高速なメモリであるが，容量が 2018 年現在の最新の GPU でも 128KB と小容量である．シェアードメモリは，32 個のバンク（Bank）と呼ばれる均等なサイズで分割されている．このバンクは，各スレッドが異なるバンクにアクセスすることで最も高速にアクセス可能となる．図

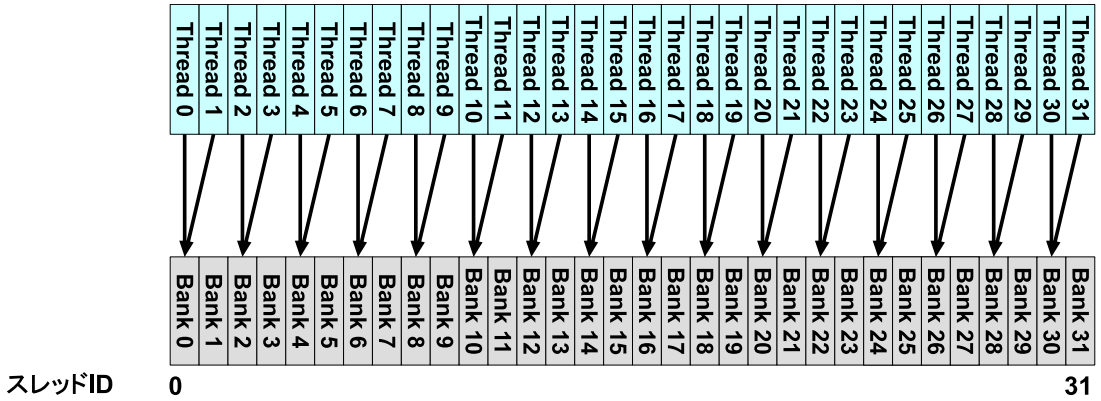

図 2–8 ： バンクコンクリフトのアクセス例

2–6 と図 2–7 に理想的なアクセス例を示す． 図 2–6，図 2–7 に示すように各スレッドは，異なるバンクにアクセスしていることから同一バンクにアクセス競合が発生していない．このようなアクセスは，並列アクセスパターンと呼ばれる一般的なアクセスパターンとなり，理想的なシェアードメモリに対する理想的なアクセスとなる．

一方，複数のスレッドが同一のバンクにアクセスする場合は，バンクコンクリフトが発生する．バンクコンクリフトは，各スレッドが同一のバンクにアクセスすることでメモリアクセスのトランザクションのリプレイが発生し，アクセスが逐次的になる．図 2–8 にバンクコンクリフトなアクセス例を示す． 図に示すように，複数のスレッドが同一バンクにアクセスを行う場合は，同時にメモリからデータを読み出すことは出来ず，メモリアクセスのリプレイが発生する．リプレイは，同時に同一バンクにアクセスするスレッド数が多い程，大きな遅延となるため，アルゴリズムの設計時に考慮することが重要となる．

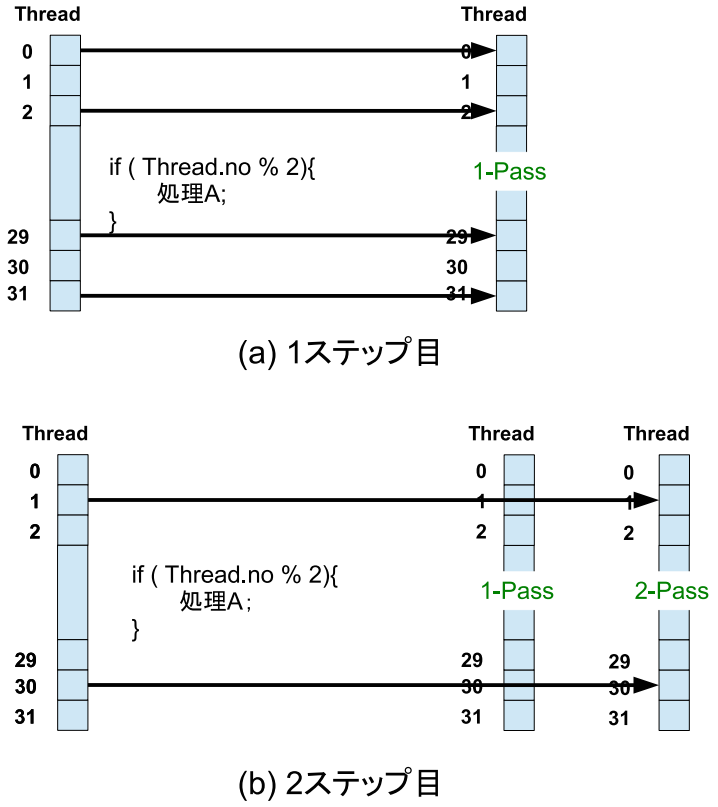

図 2–9 ： ワープダイバージェンス

2.2.3　ワープダイバージェンス

CUDA は，32 個のスレッドがワープと呼ばれる単位で纏められ実行する．このとき，ワープ内のスレッドは全て同じサイクルで同一の命令を実行する．このため，ワープ内で異なる命令を実行するスレッドがある場合は，そのワープ内の全てのスレッドが自身が実行しない命令を含めた全ての命令を実行する必要があるワープダイバージェンスが問題となる．図 2–9 にワープダイバージェンスの例を示す．図 2–9 に示すように if 文などの制御命令が必要となるプログラムを実行すると CUDA は，2 ステップに渡って命令を実行する．まず，制御命令の判定が真となる場合に実行するべき命令を実行し，次のステップで制御命令の判定が偽となる場合に実行するべき命令を実行する．最後に，必要な命令の実

行結果のみをマスクを実行して取り出す．このように，ワープダイバージェンスは制御命令の複雑性によって多くの不要な命令を実行することが考えられるため，CUDA による処理の高速化には，ワープダイバージェンスを防ぐアルゴリズムの設計が重要となる．

2.3 本章のまとめ

本章では，CUDA プログラミングにおいて重要となる GPU アーキテクチャ，CUDA プログラミングモデルについて述べた．CUDA アーキテクチャは，SIMT ホストとデバイスでメモリ空間が異なるため通信の最適化，メモリアクセスの最適化，計算スレッドが処理する命令の最適化が重要である．

第3章

メモリアクセスの局所性の向上

3.1 はじめに

本章では，格子ボルツマン法（Lattice Boltzmann Method:LBM）を題材にし，レジスタやシェアードメモリなどの階層的なメモリを用いる最適化手法について提案する．

GPU を用いた LBM で高い性能を得るためには，GPU のスレッド階層とメモリ階層を活かすようなプログラミングが必要である [21]．GPU を用いた LBM は，解析領域をブロック形状に分割してスレッドブロックに割り当て，ブロック内の格子点をスレッドに割り当てることで高い性能を得ることができる．一方で，CPU と GPU は別々のアドレス空間を持つため，解析する問題の規模が大きくなるほど，CPU と GPU 間で PCI-Express のように低速なバスを介したデータ転送が頻繁に必要となる．このため，GPU-CPU 間の通信回数を削減する手法のひとつとしてテンポラルブロッキングが提案されている [22][23][24]．

テンポラルブロッキングは，解析領域を複数のブロックに分割し，ブロック領域ごとに GPU にデータを転送し計算する手法である．本手法は，割り当てられた領域だけでなく袖領域と呼ばれる冗長な領域に対する解析も行うことで，複数の時間ステップにわたる解析をブロック領域ごとに独立に計算する．本手法を用いることで，ブロック分割による空間的局所性に加えて高い時間的局所性を得られ，データの通信回数を削減することができる．

グローバルメモリやシェアードメモリ，レジスタなどの複数のメモリ階層を持つ CUDA

を用いた LBM では，単一 GPU 内でもデータ通信が頻繁に起こる．このため，グローバルメモリ，シェアードメモリ，レジスタ間のデータ通信を削減することで LBM をより高速化できると考えられる．単一 GPU による CUDA のメモリ階層を利用してテンポラルブロッキングを行った報告のひとつに，ポアソン方程式を用いた文献 [18] があり，高い効果が得られることが報告されている．LBM においてもテンポラルブロッキングを適用することで，単一 GPU における CUDA において高速化が見込めると考えられる．

単一 GPU 内での複数のメモリ階層においてデータ通信を削減するために，LBM の計算における実行メモリバンド幅の効率化を行いワープの単位でデータに効率良くアクセスできるようにデータレイアウトを工夫する手法 [25] や，並進と衝突演算のカーネルをまとめる手法 [26] などが提案されている．これらの手法は，単一ステップにおける効率化を図るものであるため，テンポラルブロッキングを利用することで LBM をさらに高速化できると考えられる．これらを踏まえ本章では，LBM に対し単一 GPU 内のメモリ階層それぞれにテンポラルブロッキングを適用する．シェアードメモリの階層を用いるテンポラルブロッキング手法は，スレッドブロック内で計算に必要なデータをシェアードメモリへ格納する．スレッドブロック内の各スレッドは，シェアードメモリに毎ステップデータアクセスを行い計算する．このため，本手法は，1 格子点あたりの計算に必要なデータ量が多いため，シェアードメモリに格納可能な格子点数と起動可能なスレッド数の制限により，テンポラルブロッキングの段数を増やすことが難しい．一方，テンポラルブロッキングの段数を増やすことで，各スレッドが占有可能なレジスタ数を多くすることができる．このため，シェアードメモリとレジスタを用いて階層的にテンポラルブロッキングを行うことで高速化が期待できる．

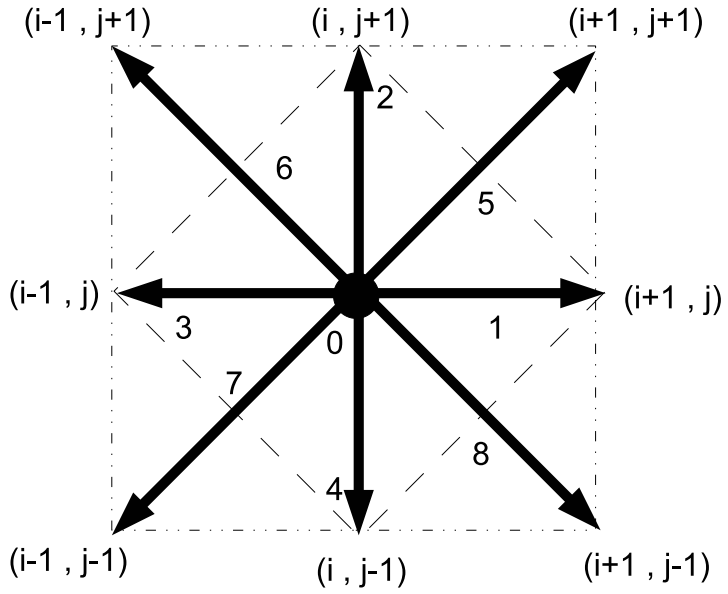

図 3–1 ： D2Q9 モデル

3.2　格子ボルツマン法

格子ボルツマン法は，解析領域を等間隔な格子で離散化し，タイムステップごとに格子上にある粒子の動きを衝突・並進の二つの分布関数を計算する子で解析する．本章では，離散化方法に D2Q9 を用いる場合の例を示す．図 3–1 に D2Q9 モデルを示す．図中の点線は格子を，格子内の矢印は方向 i に向かう速度ベクトル $\mathbf{c_i}$ を表す．本モデルは図 3–1 に示すように，粒子が 9 方向に移動するモデルである．各タイムステップの粒子の分布状態 f_i を式 (3–1) の格子ボルツマン方程式で表す．

$$f_i(\mathbf{x}+\mathbf{c_i}, t+1) = f_i(\mathbf{x}, t) + \hat{\Omega}_i[f_i(\mathbf{x}, t)](i = 0, 1, \cdots, 8) \tag{3–1}$$

式 (3–1) の右辺は，図 3–2 に示すように周囲の隣接する格子点から移動した粒子が衝突する様子を表しており，左辺は図 3–3 に示すように粒子が衝突して移動する様子を表す．式中の t はタイムステップ，$\mathbf{x}$ は位置ベクトル，$\mathbf{c_i}$ は 9 方向の方向 i に向かう速度ベクトル，

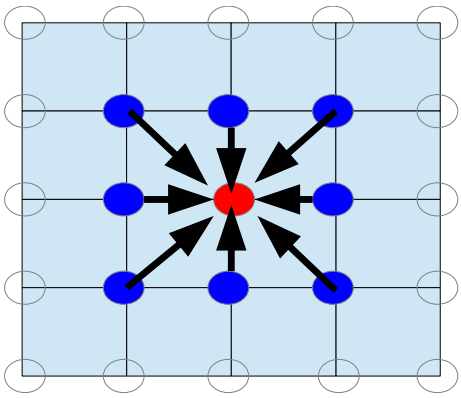

図 3–2 ： 衝突計算

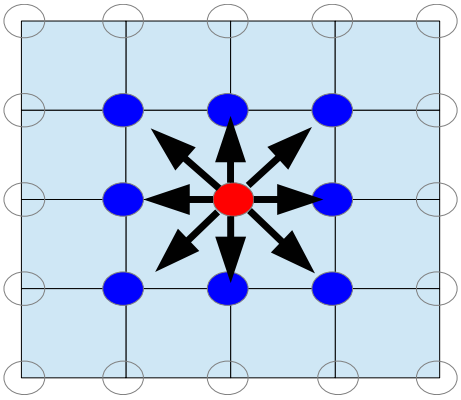

図 3–3 ： 並進計算

$\hat{\Omega}_i$ は衝突演算子である．衝突演算子 $\hat{\Omega}_i$ には，一般的に BGK 近似が用いられる [27]．BGK 近似を用いると，衝突演算子 $\hat{\Omega}_i$ は，式 (3–2) で表す．

$$\hat{\Omega}_i[f_i(\mathbf{x},t)] = -\frac{1}{\tau}[f_i(\mathbf{x},t) - f_i^{eq}(\mathbf{x},t)](i = 0, 1, \cdots, 8) \tag{3–2}$$

ここで，τ は緩和時間係数，f_i^{eq} は局所平衡分布関数である．また，密度 $\rho(\mathbf{x},t)$，流速 $u(\mathbf{x},t)$ は，それぞれ式 (3–3)，式 (3–4) で表す．

$$\rho(\mathbf{x},t) = \sum_i f_i(\mathbf{x},t) \tag{3–3}$$

$$\mathbf{u}(\mathbf{x},t) = \frac{1}{\rho(\mathbf{x},t)} \sum_i \mathbf{c_i} f_i(\mathbf{x},t) \tag{3–4}$$

また，$\mathbf{c_i}$ は，x，y 軸方向の格子点間の距離を 1 とすると式 (3–5) で表すことができる．

$$\begin{aligned} c_0 = (0,0), \quad c_1 = (1,0), \quad c_2 = (1,1), \\ c_3 = (0,1), \quad c_4 = (-1,1), \quad c_5 = (-1.0). \\ c_6 = (-1,-1), \quad c_7 = (-1,0), \quad c_8 = (1,-1) \end{aligned} \tag{3–5}$$

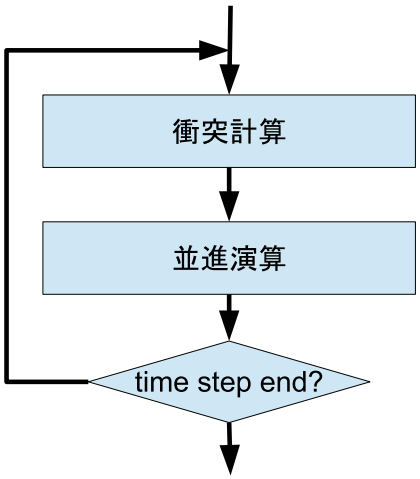

図 3–4 ： 格子ボルツマン法のフローチャート

例として $\mathbf{c_i}$ は，方向 0 ならば $\mathbf{c}_0 = (0,0)$，方向 1 ならば $\mathbf{c}_1 = (1,0)$，方向 2 ならば $\mathbf{c}_2 = (1,1)$ となる．D2Q9 モデルの局所平衡分布関数 f_i^{eq} は，式 (3–6) に示す重み係数 ω_i を用いて，式 (3–7) のように表す．

$$\omega_i = \begin{cases} \frac{4}{9} & (i = 0) \\ \frac{1}{9} & (1 \leq i \leq 4) \\ \frac{1}{36} & (5 \leq i \leq 8) \end{cases} \tag{3–6}$$

$$\begin{aligned} f_i^{eq}(\mathbf{x},t) &= \omega_i \rho(\mathbf{x},t) \left[1 - 1.5\mathbf{u}^2(\mathbf{x},t) \right. \\ &\quad \left. + 3\mathbf{c_i}\mathbf{u}(\mathbf{x},t) + 4.5(\mathbf{c_i}\mathbf{u}(\mathbf{x},t))^2\right] \end{aligned} \tag{3–7}$$

格子ボルツマン法のプログラムのフローチャートを，図 3–4 に示す．図に示すように格子ボルツマン法のプログラムは，各格子点で衝突と並進演算をタイムステップ分繰り返し計算することで求解する．格子ボルツマン法は，ボルツマン方程式を各格子点でそれぞれ計算するため並列性が高い．

3.3　CUDA を用いた格子ボルツマン法

CUDA を用いた格子ボルツマン法は，階層的なメモリ構造を効率良く用いるために，解析領域をブロック分割して各スレッドブロックに計算領域を割り当て計算する．図 3–5 に，解析領域をブロックに分割してスレッドブロックに割当てる例を示す．図に示すように，

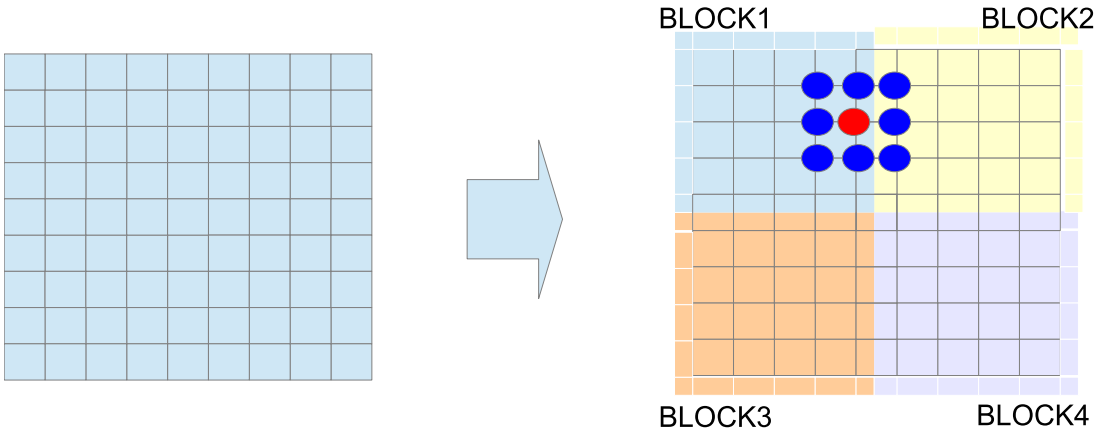

図 3–5 ： 解析領域を各ブロックに割当てる例

格子ボルツマン法の各格子点の計算は，周囲の格子点のデータが必要となる．このため，ブロックの端の格子点の計算には隣のブロックのデータが必要となる．よって，ブロック分割による LBM の計算は，スレッドブロックへ解析領域を割当てるときに，袖領域と呼ばれるブロックの一つ外側の解析点も割当てる．しかし，解析領域を割当てられる際には，必ず全てのブロックで各ステップの並進・衝突の計算が終了している必要がある．このため，CUDA による LBM の計算は，衝突・並進それぞれの計算時に必ず全てのスレッドブロックで同期処理が必要となる．図 3–6 に，CUDA による格子ボルツマン法のフローチャートを示す．

スレッドブロックは，割り当てられた計算領域をシェアードメモリに書き込み 1 ステップ分計算し，結果をグローバルメモリに書き戻す．本手法は，シェアードメモリを用いる

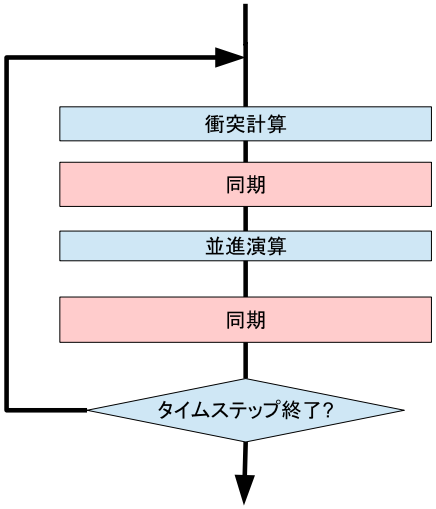

図 3–6 ： CUDA による格子ボルツマン法のフローチャート

ことで空間的局所性が得られるが，毎時間ステップごとにグローバルメモリへアクセスするため，メモリアクセスコストが高い．

3.3.1　テンポラルブロッキング

テンポラルブロッキングは，マルチコア環境など複数のメモリを持つ実行環境においてステンシル計算を高速化するために提案された手法である．格子ボルツマン法をはじめとするステンシル計算は，メモリ領域に収まらないような解析領域の計算を行う際には，解析領域を複数のブロックに分割して各メモリに割当てて計算する．このとき，ブロックの端の領域は，袖領域と呼ばれ隣接ブロックとの計算結果の通信が必要となる．隣接ブロックとの通信は，計算コストに比べ相対的に大きくなるため毎時間ステップごとに袖領域の通信を行うと通信にかかるオーバーヘッドが無視できなくなる．

この問題を解決する手法として，テンポラルブロッキングが提案されている．本手法は，ブロック分割による空間的局所性だけでなく，通信に必要な袖領域の幅を増やすことで，

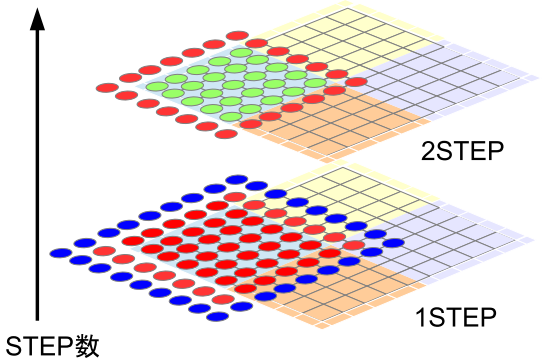

図 3–7 ： 段数 2 段のテンポラルブロッキング

隣接ブロックとの通信を行わずに複数時間ステップの計算を可能にする時間的局所性の効果を得ることができる．本章では，1 ステップを 1 段として定義し，図 3–7 に 2 段のテンポラルブロッキングの例を示す．図に示すように，自身の計算が必要になるブロックよりも，2 段分の計算に必要な袖領域を割当てる．1 段目の計算では，一番外側の袖領域を使い計算を行う．2 段目の計算では，一番外側の袖領域は使わずに一つ内側の袖領域を使い計算を行う．このように，計算を行うことで毎段数ごとに袖領域の通信を行わなくても，複数段数の計算が可能となる．一方で本手法は，袖領域を段数分読み込み袖領域の計算も行う．このため，隣接するブロックが計算するはずの袖領域を重複して計算するため，無駄な計算を行う．

3.4　メモリアクセスコストを削減するテンポラルブロッキング手法

提案手法は，CUDA の各メモリ階層においてテンポラルブロッキングを適用することで LBM を高速化する．CUDA を用いた LBM を高速化するために，シェアードメモリを用いたテンポラルブロッキング（STB），レジスタを用いたテンポラルブロッキング（SRTB）

による最適化を行う．提案手法のテンポラルブロッキングのフローチャートを図 3–8 に示す．本例は，STB の段数が ts 段，SRTB の段数が tr 段における例を示しており，図中の $timesteps$ はシミュレーション時間，ts はシェアードメモリのテンポラルブロッキング段数，tr はレジスタのテンポラルブロッキング段数を格納する．提案手法は，CPU 上で解析時間を $timesteps$ で制御し，GPU 上で LBM の衝突並進を計算する．LBM の衝突並進演算を行う CUDA カーネルは，STB ループと RTB ループの 2 重ループで構成される．STB ループは，シェアードメモリのテンポラルブロッキングの段数分ループする．また，RTB ループは，レジスタのテンポラルブロッキングの段数分ループする．このため，本例における $tr = 1$ の時は，STB の動作となる．各スレッドが割り当てられた格子点における衝突並進は，RTB ループ中で計算する．STB ループで各スレッドに割り当てられる計算に必要なデータは，グローバルメモリから読み込むため，STB ループの外で一度のみシェアードメモリへ格納する．これにより，グローバルメモリへのメモリアクセスコストが削減できる．RTB ループで各スレッドが割り当てられた格子点の計算に必要なデータは，シェアードメモリからロードするため，RTB ループの外でレジスタにデータを格納する．これにより，RTB ループ内では STB ループで必要になる同期処理を行うことなく複数段の計算が可能となる．また，各格子点の衝突並進演算を行う RTB ループの段数 tr は，シェアードメモリのテンポラルブロッキングの段数 ts を超える段数を指定すると袖領域のデータがないため指定できない．このため，各テンポラルブロッキングの段数の指定は $tr \leqq ts$ となる．

以下では，STB，SRTB について述べ，更に SRTB による冗長な計算を削減する手法について述べる．

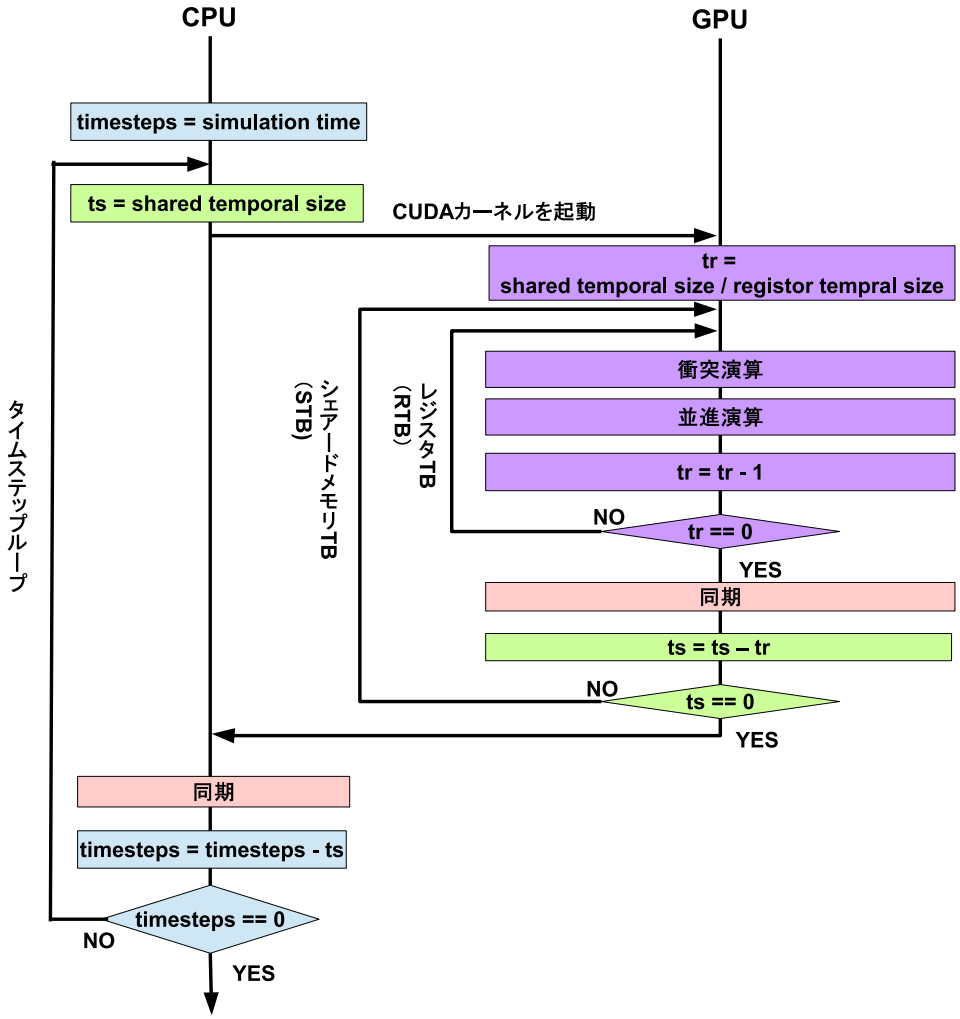

図 3–8 ： RTB のループ段数 ts，SRTB のループ段数 tr によるテンポラルブロッキングのフローチャート

3.4.1　シェアードメモリを用いたテンポラルブロッキング手法

本手法は，ブロック分割した領域よりも広い領域を低レイテンシでアクセスできるシェアードメモリに格納する．図 3–9 に，解析領域をシェアードメモリに割り当ててテンポラルブロッキングにより計算する処理の流れを示す．本手法は，まず，ステップ毎に結果を格納する領域を切り替える必要があるため，シェアードメモリに 2 個のバッファ領域を確

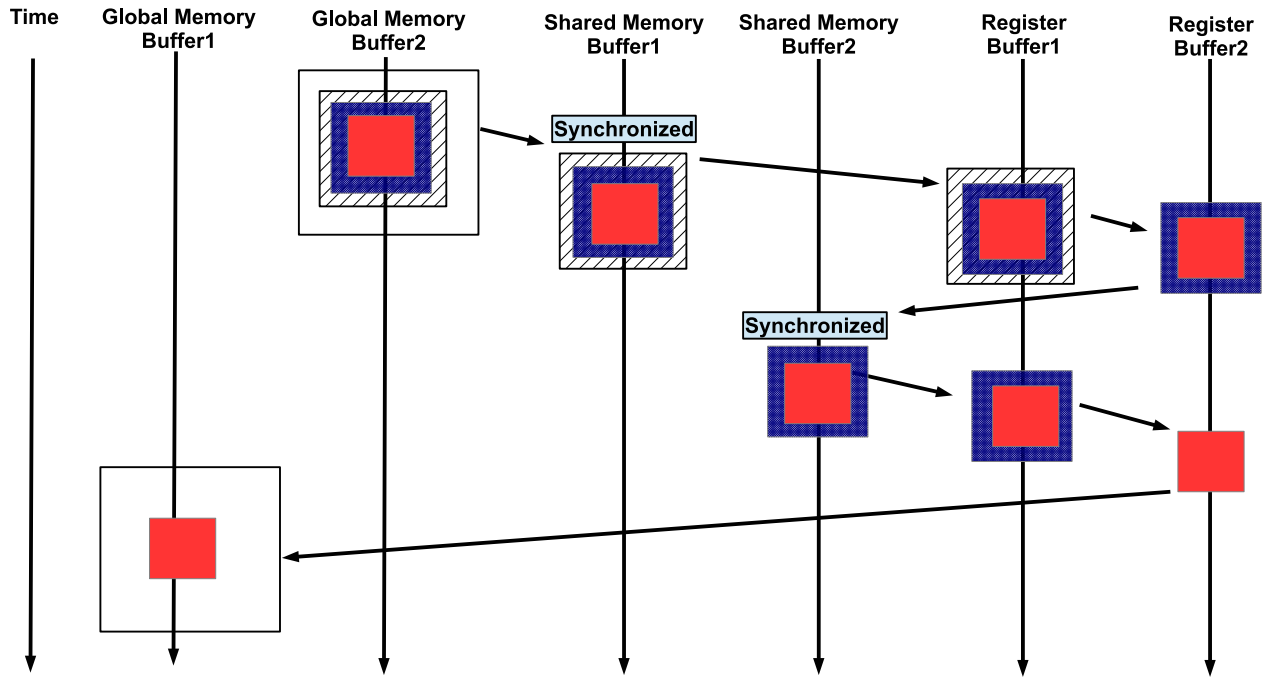

図 3–9 ： STB によるテンポラルブロッキング 2 段の計算例

保する．次に，スレッドブロックは，グローバルメモリから計算に必要な袖領域を含む格子点データをシェアードメモリのバッファ領域 1 に格納する．割り当てる際に更新される要素は，グローバルメモリ上でハッチングされているエリアである．次に，シェアードメモリのバッファ領域 1 からレジスタに格子点データをロードし，レジスタ上で格子点データを更新する．このとき，他のスレッドがシェアードメモリのバッファ領域 1 からデータを読み込むことがあるため，更新が終了するとシェアードメモリのバッファ領域 2 へ更新したデータを格納する．次の時間ステップでは，シェアードメモリのバッファ領域 2 のデータをレジスタにロードして格子点情報を計算し，計算が終了すると更新データをシェアードメモリのバッファ領域 1 へ格納する．本手法は，テンポラルブロッキングの段数が多くなるほどシェアードメモリへのアクセスを繰り返すため，メモリアクセス遅延が発生する．

3.4.2　レジスタを用いたテンポラルブロッキング手法

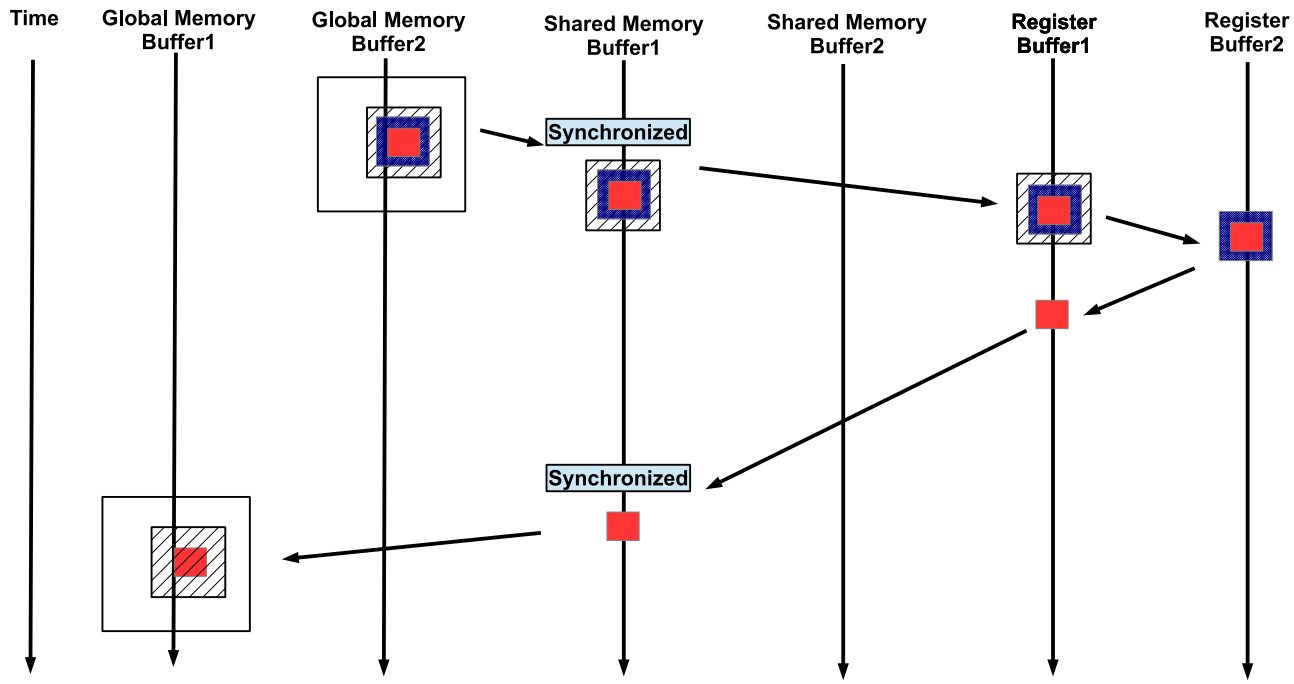

図 3–10 ： SRTB によるテンポラルブロッキング 2 段の計算例

シェアードメモリのアクセス遅延を最小限にするために，シェアードメモリ上で行われるテンポラルブロッキングをレジスタ上で行う．本手法は，各スレッドブロックで用いるシェアードメモリへのアクセスは計算カーネルの初回のみ行い，各スレッドが計算に必要なデータをそれぞれレジスタに格納する．このため，本手法は，シェアードメモリによる手法よりも多くのレジスタを必要とする．図 3–10 に，提案手法で 2 段のテンポラルブロッキングを計算する流れを示す．レジスタを用いたテンポラルブロッキングは，まず，初回のグローバルメモリから分割された領域データをシェアードメモリ上に袖領域を含めてロードする．次に，レジスタ上でテンポラルブロッキングを行うために，レジスタ上にバッファ領域 1 と 2 を確保する．各スレッドが計算するレジスタのデータは，他のスレッド間で参照，格納ができないため，各スレッドは自身が計算する領域のデータを自身が操

作できるレジスタに全て格納する．各スレッドは，レジスタ上のバッファ領域 1 のデータを用いて 1 段目を計算し，1 段目の計算結果をバッファ領域 2 に格納し，バッファ領域 2 を用いて 2 段目を計算する．2 段目の計算が完了したら，グローバルメモリへ自身の計算した領域のデータを格納する．これにより，時間ステップが進む際に必要になる他スレッドの計算結果を参照せずに計算可能であり，同期コストも削減できる．

3.5　評価

GPU を用いた格子ボルツマン法に対するレジスタを用いたテンポラルブロッキングの有効性を確認するために，2 次元ポアズイユ流れ [28] を解析する．評価環境は，CPU が Intel Xeon E5-2687W，GPU が Titan X Pascal である．表 3–1 に，評価環境の構成を示す．本評価で解析するポアズイユ流れのモデルは，D2Q9 モデルで一辺 320 格子に離散化し，解析領域の上下の境界条件を bounce-back 条件 [29]，解析領域の左右の境界条件を周期境界条件とする．また，評価に用いたプログラムでは，各格子点のデータを x 座標優先で単精度の 1 次元配列に格納する．

3.5.1　STB と SRTB の実行時間の評価

テンポラルブロッキングに対してレジスタを利用する有効性を確認するために，シェアードメモリを用いたテンポラルブロッキングとレジスタを用いたテンポラルブロッキングの実行時間を測定する．図 3–11 に，シェアードメモリとレジスタを用いたテンポラルブロッキング段数とブロック数によるポアズイユ流れの実行時間を示す．図中では，シェアードメモリのテンポラルブロッキングが n 段，レジスタのテンポラルブロッキングが m 段の手法を snr$m$tb と表記する．つまり，s2r1tb は，シェアードメモリ上で 2 段，レジスタ上で 1 段のテンポラルブロッキングを行うことを表す．また，レジスタを用いないシェ

表 3–1 ： 評価環境

CPU	Processor	Intel Xeon E5-2687W
	Memory	32GB
GPU	Processor	Nvidia Titan X Pascal
	Global Memory	12GB
	Shared Memory	48kB
	L1 cache	16kB
	L2 cache	3MB
	CUDA core	3584
	CUDA Version	CUDA 9.0

アードメモリのテンポラルブロッキングの段数での表記は，レジスタ段数を全て 0 とする．つまり，s0r0tb はテンポラルブロッキングを適用しない手法を表す．

本評価の測定条件では，n の設定を 8 以上にするとシェアードメモリの容量不足により実行不能となる．同様に，s4r0tb，s8r0tb，s2r1tb，s4r1tb，s8r1tb も実行不能であるため，図中では測定結果を空欄とする．

図 3–11 より，全てのブロックサイズの条件においてテンポラルブロッキング段数が増加するごとに，処理時間が短縮することが分かる．ブロックサイズ 10×10 のとき，シェアードメモリを用いた s8r0tb は，テンポラルブロッキングを適用していない s0r0tb に対して，最も高い約 7.36 倍の高速化が得られることが確認できた．これは，グローバルメモリへのアクセスコストが減少したことや，アクティブブロック数を複数起動することによる，グローバルメモリやシェアードメモリのアクセス時間隠蔽の効果であると考えられる．

メモリアクセスの隠蔽による効果を確認するために，NVVP[30] を用いて s8r0tb と s0r0tb

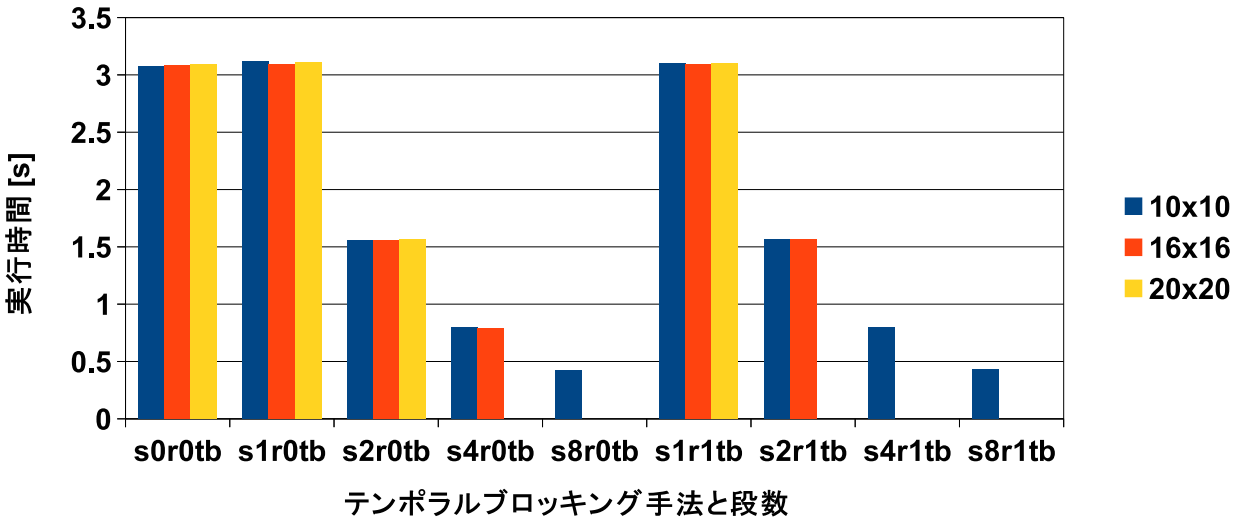

図 3–11 ： 実行時間

表 3–2 ： メモリアクセスのストールの割合（%）

	ストールの割合
s0r0tb	6.0
s8r0tb	0.3

において全体の処理時間に占めるメモリ操作に要した割合とアクティブブロック数を測定する．まず，表 3–2 に，処理時間に占めるメモリアクセスのストールの割合を示す．表 3–2 より，s8r0tb と s0r0tb における全体の処理時間に占めるメモリアクセスのストールの割合が s0r0tb が約 6%，s8r0tb が約 0.3%であることが確認できる．これは，メモリアクセスコストの低いシェアードメモリ上で複数ステップの計算を行い，グローバルメモリへのアクセスが削減できたことによる効果であることが分かる．次に，図 3–12 に，各手法におけるアクティブブロック数を測定した結果を示す．図 3–12 より，s0r0tb におけるア

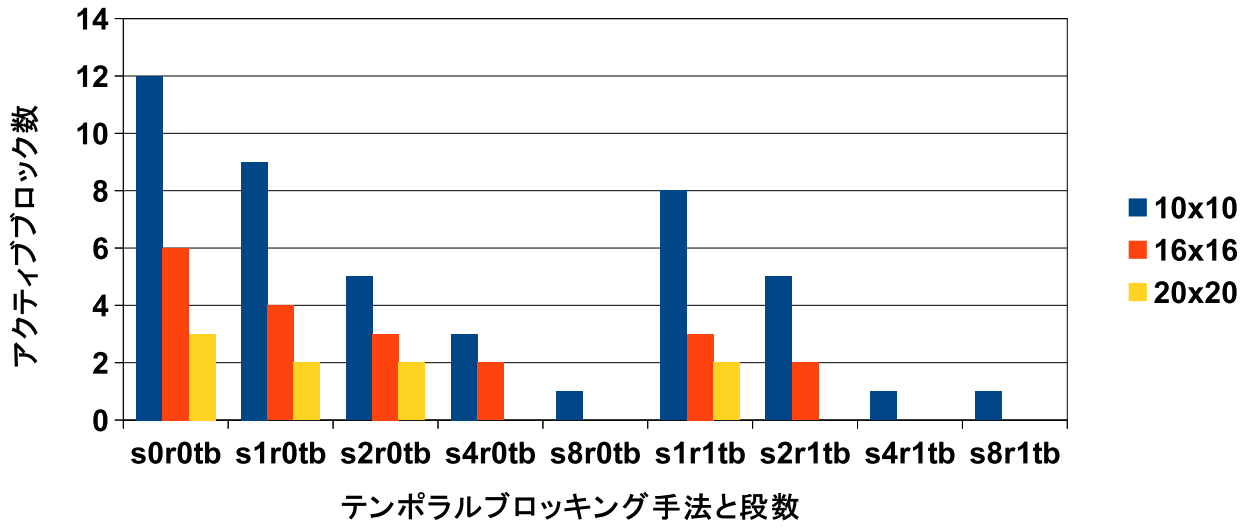

図 3–12 ： アクティブブロック数

クティブブロック数が最も多く，段数が増加するとアクティブブロック数が減少することがわかる．また，図 3–11 において，高速な手法ほどアクティブブロック数が低いことがわかる．通常，CUDA は，高いメモリアクセスコストを隠蔽して処理を高速化するために，多くのアクティブスレッドブロック数を確保することが重要である．しかし，LBM は，格子計算の中でも特に計算に必要なメモリのコストが高いメモリバウンドな手法である．このため，テンポラルブロッキング段数を増やすとスレッドブロックが計算に必要とするデータ量が多くなり，アクティブブロック数が減少したと言える．アクティブスレッド数が低いにも関わらず高速化した要因は，メモリアクセスにかかる割合と計算の割合が関係すると考えられる．そこで，s8r0tb と s8r1tb の計算の割合と，メモリアクセスによる処理のストールの割合を NVVP で測定する．表 3–3 に，NVVP で測定した結果を示す．表 3–3 より，s8r1tb の計算の割合は約 80%，メモリアクセスの割合が約 1%であるのに対し，s8r0tb の計算の割合は約 70%，メモリアクセスの割合が約 3%である．このため，メ

表 3–3 ： 計算とメモリアクセスの割合（%）

	計算の割合	メモリアクセスの割合
s8r0tb	70	3.0
s8r1tb	80	1.0

モリアクセスにかかる時間を減らし計算の割合が増加したことで，計算が高速化できたと考えられる．

3.6　本章のまとめ

本章では，CUDA を用いた格子ボルツマン法を高速化するために，シェアードメモリ，レジスタ上でテンポラルブロッキングを適用する手法について提案した．評価の結果，シェアードメモリを用いた s8r0tb 手法は，s0r0tb 手法に比べて最大約 7.36 倍の高速化を得ることが確認できた．

第4章

並列性の抽出による高速化

4.1 はじめに

本章では，拡張ベクトル化LU分解法を題材に，GPUの実行単位であるワープに最適化する形で並列性を抽出する手法を提案する．GPUを用いた拡張ベクトル化LU分解法の高速化には，GPUのSIMTである実行形式を十分に考慮した並列性の抽出が必要である．

拡張ベクトル化LU分解法は，電子回路や電力計算などの分野[31]で求解が必要となるランダムスパース方程式求解を解くための直接法による求解手法のひとつである．ランダムスパース方程式は，約90％以上が零要素となるスパース性の高い行列であり，並列性の抽出が困難な方程式である．このため，従来より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において並列性を抽出する手法として，行列をブロックに分割する手法[32][33]や列レベルのタスクスケジューリングを行う手法[34][35]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，列同士のデータ依存を解析して同時に実行可能な列同士を組み合わせ，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．

そこで，超並列なベクトルプロセッサ向けに命令レベル並列性を利用する拡張ベクトル

化 LU 分解法などの並列性の抽出手法が提案されている [31][36]．本手法は，処理に必要な全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．しかし，拡張ベクトル化 LU 分解法で生成するベクトル命令は，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．このような命令の組み合わせを GPU で処理する場合，ワープダイバージェンスによる実行効率の低下が起こり，処理速度が低下する．そこで，本章では，ランダムスパース方程式を GPU で効率よく求解するために，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スを防止する手法を提案する．

4.2　ランダムスパース方程式求解手法

本章では，連立一次方程式の求解について述べる．連立一次方程式は，多くの物理特性を表す問題を方程式にモデル化した際に解く必要がある．また，これらの物理特性を表す問題は，非線形連立方程式の求解が必要となることから，何回も同じ方程式を求解するため，求解の高速化が求められている．連立一次方程式を解く手法に，直接法と反復法がある．反復法は，物理特性を表す問題を方程式にモデル化したとき，帯行列のように特定の係数行列の形を持つ場合，帯行列という特徴を利用した高速に求解可能な反復法が利用できる．しかし，本章で扱うランダムスパースな係数を持つ連立一次方程式の求解は，非ゼロ要素の位置がランダムであるため，反復法による求解を用いる場合でも密行列として扱う必要がある．

以上の問題により，ランダムスパースな係数を持つ方程式求解には，直接法による求解が行われる．直接法は，必ず有限解の演算で解が得られるという特徴がある．直接法の一つであるガウスの消去法は，与えられた連立方程式から一つの一次方程式を選び，その方

程式以外の一次方程式を用いて連立一次方程式の消去計算を行うことで元数を一つ減らす．そして，更に元数を一つ減らした方程式同士で同様の処理を繰り返し行う．このようにして，計算を行った一元化方程式を用いて元数の一つ多い方程式の解を求める．この手法は，通常連立一次方程式を手計算で求めるように方程式を求解する．

そして，ガウスの消去法を元に計算量の削減を行った手法に LU 分解法がある [37]．本章でも扱うランダムスパースな係数を持つ連立一次方程式の場合も適用できるが，スパース性の高い行列を LU 分解法による求解を行うと，係数行列中のゼロ要素が求解処理によって非ゼロ要素に変化する fill-in の発生により計算量が増加する．このため，リオーダリングを行うことで計算量の削減ができる [37][38].

本章では，以下 LU 分解法と Markowitz 法によるリオーダリング手法について述べる．

4.2.1　LU 分解法

LU 分解法は，厳密解法である直接法の一つである．直接法は，有限解の演算で厳密解が得られることができるが，次数が大きいと計算量が膨大になり求解時間が大きくなる．

以下では，LU 分解法について述べる．次式 (4–1) が与えられたとする．

$$\begin{array}{ccccccccc} a_{11}x_1 & + & a_{12}x_2 & + & \ldots & + & a_{1n}x_n & = & b_1 \\ a_{21}x_1 & + & a_{22}x_2 & + & \ldots & + & a_{2n}x_n & = & b_2 \\ \vdots & & \vdots & & & & \vdots & & \vdots \\ a_{n1}x_1 & + & a_{n2}x_2 & + & \ldots & + & a_{nn}x_n & = & b_n \end{array} \tag{4–1}$$

この連立一次方程式を，次のような行列に置き換えることができる．

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \dots & a_{nn} \end{bmatrix} \qquad x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \qquad b = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{4–2}$$

式 (4–1) は，式 (4–2) ように置き換えることができ，行列の形として連立一次方程式の式は，次式 (4–3) と表せる．

$$Ax = b \tag{4–3}$$

LU 分解法では，連立一次方程式 $Ax = b$ を解くにあたり，係数行列 A を次式 (4–4) のように下三角行列 L と上三角行列 U とに分解する．

$$A = LU \tag{4–4}$$

各行列，L と U は以下のようになる．

$$L = \begin{bmatrix} l_{11} & & & 0 \\ l_{21} & l_{22} & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} \dots & l_{nn} & \end{bmatrix} \qquad U = \begin{bmatrix} 1 & u_{12} & \dots & u_{1n} \\ & 1 & & u_{2n} \\ & & 1 & \vdots \\ 0 & & & 1 \end{bmatrix} \tag{4–5}$$

このように，係数行列 A を L と U に分解することを LU 分解という．

次に，代入計算である前進後退を行うことで解 x を求める．式 (4–3) を LU 分解した式 (4–4) は，次式 (4–6) が成り立つことがわかる．

$$Ax = LUx \tag{4–6}$$

式 (4–6) は，右辺を変形させることで次式 (4–7)(4–8) という方程式で表すことができる．

$$Ly = b \tag{4–7}$$

$$Ux = y \tag{4–8}$$

式 (4–7) の手順を前進代入といい，式 (4–8) の手順を後退代入という．これら式 (4–7) と式 (4–8) を解くことで解 x を求める．

式 (4–7) は次の行列計算のように表すことができる．

$$\begin{bmatrix} l_{11} & & & 0 \\ l_{21} & l_{22} & & \\ \vdots & \vdots & \ddots & \\ l_{n1} & l_{n2} & \dots & l_{nn} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{4–9}$$

まず $l_{11}y_1 = b_1$ の方程式を解く．この場合，l_{11} と b_{11} は既に定まっているので y_1 が求まる．次に，y_2 を求めるために，$l_{21}y_1 + l_{22}y_2 = b_2$ の場合も，l_{21}，l_{22}，b_2 は既に定まっていて，y_1 も先の計算で求まっているので，y_2 が求まる．以下同じ用にこのような処理を繰り返すことで，y を求めることができる．このようにして得た y を用いて，式 (4–8) の後退代入の処理を行い，解 x を求めることができる．

式 (4–8) は次の行列計算として表すことができる．

$$\begin{bmatrix} 1 & u_{12} & \dots & u_{1n} \\ & 1 & \dots & u_{2n} \\ & & 1 & \vdots \\ 0 & & & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{4–10}$$

後退代入では，$ux = y$ の計算を行う．この時 y_n は既に求まっているので x_n が求まる．次に x_{n-1} に対して，同じように x_1 まで繰り返し行うことで，解 x を求めることができる．

LU 分解法では，以上のように LU 分解である分解計算と代入計算である前進後退代入が分かれている．LU 分解法は，分解計算を一度行うことで，右辺を変更するときには代入計算のみでよい．分解計算にかかる計算量は $O(n^3)$ であるのに対して，代入計算にかかる計算量は $O(n^2)$ である．このため，右辺のみの計算を行う際には，計算量が大幅に削減できる．また，LU 分解法の分解計算では，$O(n^3)$ の計算量がかかるため，処理の高速化のために，様々な分解アルゴリズムが提案されている．この手法の一つにクラウト法がある．

4.2.2　クラウト法

クラウト法は，内積計算により分解を行う手法である．演算の方法は，係数行列 A を次式 (4–11) のように分解する．

$$
\begin{aligned}
A &= L_1 A_1 U_1 \\
&= L_1 L_2 A_2 U_2 U_1 \\
&\dots \\
&= L_1 L_2 \dots L_n A_n U_n \dots U_2 U_1 \\
&= LDU, L = L_1 L_2 \dots L_n, D = A_n, U = U_1 U_2 \dots U_n
\end{aligned} \tag{4–11}
$$

ここで，D は対角行列であり次式 (4–12) となる．

$$
D = \begin{bmatrix}
D_{11} & & & 0 \\
& D_{22} & & \\
& & \ddots & \\
0 & & & D_{nn}
\end{bmatrix} \tag{4–12}
$$

各 L_k は下三角行列であり，各 U_k は上三角行列で次式 (4–13) のようになる．

$$
L_k = \begin{bmatrix}
1 & & & & & \\
& \ddots & & & & 0 \\
& & 1 & & & \\
& & & 1 & & \\
& & & l_{k+1k} & 1 & \\
0 & & & \vdots & & \ddots \\
& & & l_{nk} & & 1
\end{bmatrix}, U_k = \begin{bmatrix}
1 & & & & & & \\
& \ddots & & & & & 0 \\
& & 1 & & & & \\
& & & 1 & u_{kk+1} & \dots & u_{kn} \\
& & & & 1 & & \\
0 & & & & & \ddots & \\
& & & & & & 1
\end{bmatrix} \tag{4–13}
$$

L_k と U_k は第 k 列ないし第 k 行だけが単位行列と異なっている．ここで，積 $L_1 \ldots L_k$ と積 $U_k \ldots U_1$ は次式 (4–14)，(4–15) の性質をもつ．

$$L_1 L_2 \ldots L_k = L_1 + L_2 + \ldots + L_k - (k-1)I \tag{4–14}$$

$$U_k U_{k-1} \ldots U_1 = U_k + U_{k-1} + \ldots + U_1 - (k-1)I \tag{4–15}$$

クラウト法のアルゴリズムを数式表現すると次式 (4–16) のようになる．

$$\begin{aligned} A = A_0 = \begin{bmatrix} a_{11} & u_1^T \\ l_1 & A(1) \end{bmatrix} &= \begin{bmatrix} 1 & 0 \\ l_1/l_{11} & I_{n-1} \end{bmatrix} \begin{bmatrix} l_{11} & 0 \\ 0 & B_1 \end{bmatrix} \begin{bmatrix} 1 & u_1^T/l_{11} \\ 0 & I_{n-1} \end{bmatrix} \\ &= \begin{bmatrix} l_{11} & u_1^T \\ l_1 & l_1 u_1^T/l_{11} + B_1 \end{bmatrix} \\ &= L_1 A_1 U_1 \end{aligned} \tag{4–16}$$

これにより，$l_{11} = a_{11}$，$B_1 = A^{(1)} - l_1 u_1 / l_{11}$ である．次に A_1 の分解を行う．

$$\begin{aligned} A = \begin{bmatrix} l_{11} & 0 \\ 0 & B_1 \end{bmatrix} &= \begin{bmatrix} l_{11} & 0 & 0 \\ 0 & l_{22} & u_2^T \\ 0 & l_2 & A^{(2)} \end{bmatrix} \\ &= \begin{bmatrix} 1 & & 0 \\ 0 & 1 & \\ 0 & l_2/l_{22} & I_{n-2} \end{bmatrix} \begin{bmatrix} l_{11} & & \\ & l_{22} & \\ & & B_2 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & u_2^T/l_{22} \\ 0 & 0 & I_{n-2} \end{bmatrix} \\ &= L_2 A_2 U_2 \end{aligned} \tag{4–17}$$

クラウト法のアルゴリズムを図示すると図 4–1 のようになる．

クラウト法は，左部と上部を参照しながら分解する手法である．

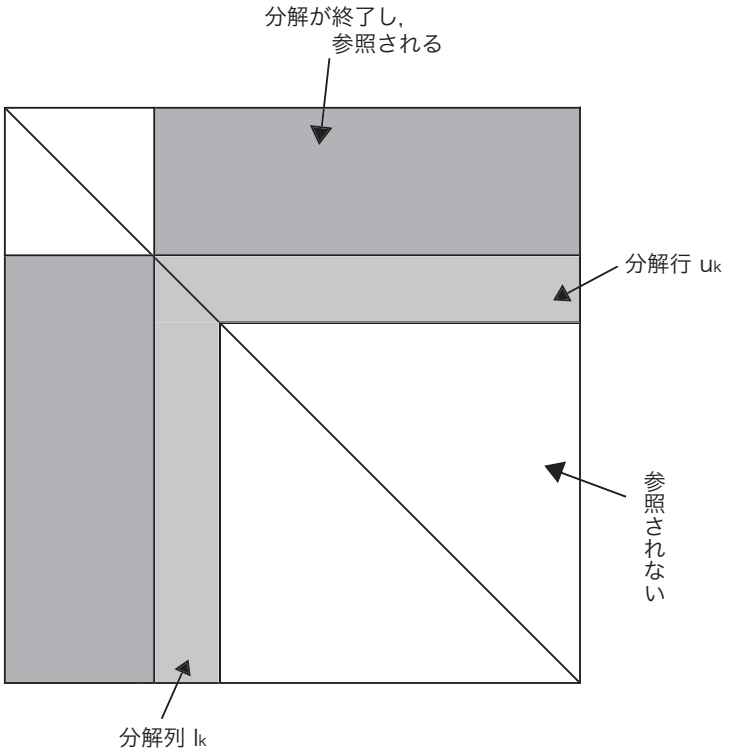

図 4–1 ： クラウト法

4.2.3　Markowitz 法のリオーダリング手法

数値計算の分野では，物理問題を方程式にモデル化を行った場合，帯行列などの特殊な構造を持ったスパース行列 (疎行列) となり，係数行列の値を変更した連立一次方程式を繰り返し方程式を解くことがある．電子回路シミュレーションや電力潮流計算などの分野では，高いスパース率の係数行列を持つ方程式を解く必要がある．ゼロ要素が多く含まれるスパース行列を LU 分解による求解を行うと，求解中にゼロ要素から非ゼロ要素に変化する fill-in が多く起こる．fill-in が増加すると，非ゼロ要素数が増加し求解に必要な計算量が増えることで処理時間が増大する [37]．このため，演算に必要な処理全体を通じて計算量または計算時間の削減が重要となり，前処理として係数行列の構造を変化させ全

体の計算量を少なくすることが行われている．このような処理をリオーダリングという [37][38]．前処理として行列の形を変化させ fill-in の数を減少させることは，係数行列がスパース行列となる場合，よく行われている．本稿では，ランダムスパースな係数行列を持つ連立一次方程式を対象としているため，前処理としてリオーダリング手法の一つである Markowitz 法によるリオーダリング手法を用いる．

Markowitz 法は，二つのフェーズにより構成される．行列中の行の要素数を行カウント，列の要素数を列カウントとして用意し，行カウントが 1 のものを左上の軸位置に持っていき，次に列カウントが 1 のものを左上の軸位置に持ってくる．このことを，核の生成という．核の生成例を，図 4–2 に示す．

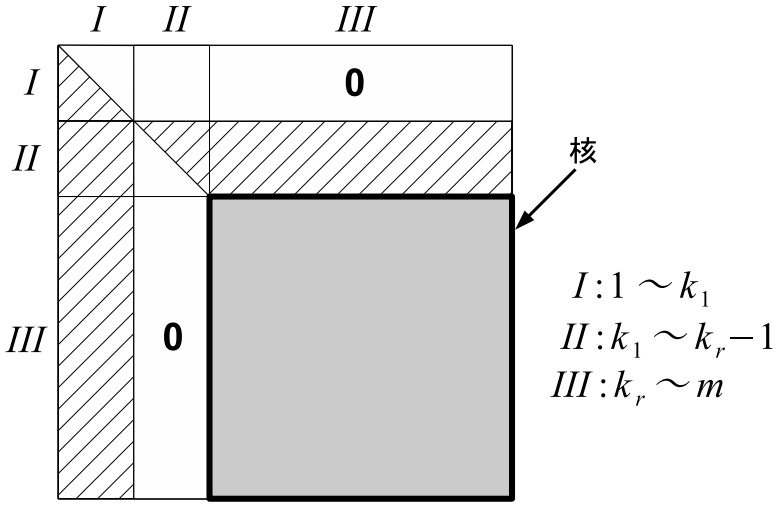

図 4–2 ： 核の生成

核の生成が終了したら，(行カウント-1)×(列カウント-1) が最小となる行 r と列 c を選んで軸位置に持ってくる Markowitz 法を行う．以下に，核の生成と Markowitz 法の手順を示す．

1. 各行と各列の非ゼロ要素の個数を数える．また，k=1 とする．

2. 行カウントが 1 の行 r を見つけ，この行の k 列目以降に存在する非ゼロ要素を a_{rc} とする．

3. r 行目と c 行目を入れ替え，k 列目と c 列目の入れ替えを行う．

4. 各行の行カウントのうち，$a_{ic} \neq 0$ となる行 i について行カウントの値から 1 を引く．

5. k に 1 を加え，行カウントが 1 の行が存在するか確認する．このとき，行カウントが 1 のものがあれば，手順 1 に戻り，なければ手順 6 に進む．

6. 列カウントが 1 の列 ν を見つけ，この行の k 行目以降に存在する非ゼロ要素を $a_{\mu\nu}$ とする．

7. k 行目と r 行目を入れ替え，k 列目と ν 列目の入れ替えを行う．

8. 各行の列カウントのうち，$a_{\mu j} \neq 0$ となる列 j について列カウントの値から 1 を引く．

9. k に 1 を加え，列カウントが 1 の列が存在するか確認する．このとき，行カウントが 1 のものがあれば，手順 6 に戻り，なければ手順 8 に進む．
   ここまでが，核の生成手順となる．

10. (行カウント-1)×(列カウント-1) が最小となる行 r と列 c を選んで軸位置，k 行目と r 行目を入れ替え，k 列目と c 列目の入れ替えを行う．このとき，k が k=行列サイズになるまで繰り返し行い，条件を満たさない場合は，処理手順 10 を繰り返し，満たした場合は，Markowitz 法を終了する．

以上のようにして，Markowitz 法によるオーダリングを行うことで，係数行列を入れ替えることにより，fill-in の数を減らし処理量の削減をする．リオーダリングによる処理を行わない場合の fill-in と，処理を行った場合の fill-in についてそれぞれ図 4–3 と図 4–4 に示す．

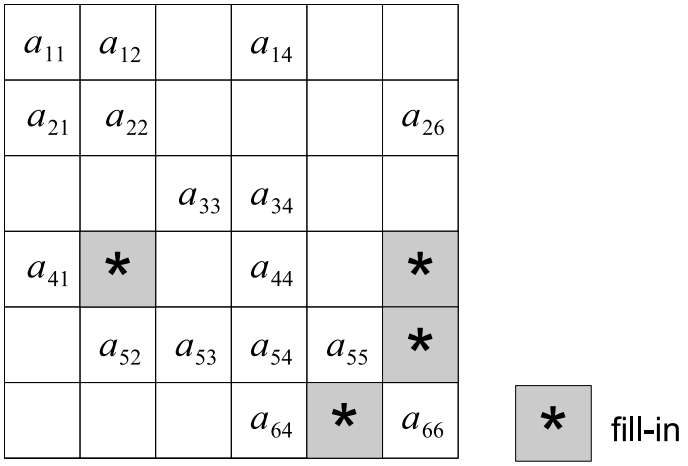

図 4–3 ： Crout 法実行時に起こる行列中の fill-in

図 4–3 は，Markowitz 法のリオーダリングを適用せずにクラウト法による LU 分解を行った場合に現れる fill-in について示している．図 4–4 は，Markowitz 法のリオーダリングを適用しクラウト法による LU 分解を行った場合に現れる fill-in について示している．行列中に空白となっている部分は，ゼロ要素が格納されており,「*」が書き込まれている部分は，クラウト法による LU 分解を行った際に fill-in が発生していることを表す．図 4–3 では，fill-in が三つ現れているのに対し，図 4–4 では，fill-in が一つしか現れていない．このように，行列の要素位置を Markowitz 法により変化させることで fill-in の発生個数を抑えることができる．

以上のように，連立方程式が与えられた際に前処理としてリオーダリングを行うことで，求解に必要な計算量を削減し，処理時間を減少させることができる．

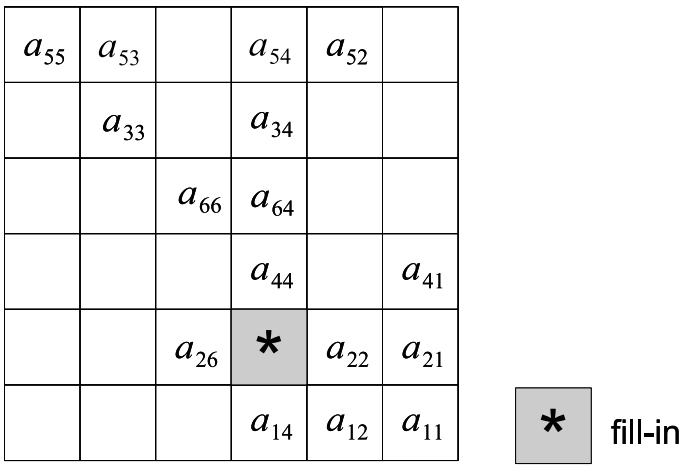

図 4–4 ： Markowitz 法適用後の行列中の fill-in

4.3　命令レベル並列性を利用したランダムスパース方程式求解の高速化

HPC の分野などで求解が行われる大規模な数値計算などは，ベクトルコンピュータなどのスーパーコンピュータを用いた方程式求解が行われている．このため，高速化のためには同時実行可能な箇所の抽出，ベクトル化が重要になる．しかし，本章でも題材としている電子回路シミュレーションなどではランダムスパースな係数行列を持つ連立一次方程式の求解を行う必要がある．このため，電子回路シミュレーションなどのゼロ要素を多く占めるランダムスパース行列を解く場合，非常に並列度が低く十分なベクトル長を確保することが困難となる．

そこで，本章ではランダムスパース行列を持つ連立一次方程式のベクトル化抽出手法に有効とされている MVA（Maximum Vectorized Algorithm）[36] と MVA を拡張した拡張 MVA（拡張レベル付けベクトル化 LU 分解法）[31] について述べる．

4.3.1　Maximum Vectorized Algorithm

LU 分解は，二種類の演算で計算することができる．このため，MVA 手法では二種類の演算を要素の参照要素に基づいてレベルという概念を利用し，同時実行可能箇所のベクトル抽出を行う．

LU 分解時の二種類の演算で用いる各要素は次 (4–18) のように表すことができる．

$$\begin{cases} \text{更新要素}\,[a] = \text{参照要素}\,(\text{更新要素})[a] - \text{参照要素}\,[b] \times \text{参照要素}\,[c] \\ \text{更新要素}\,[a] = \text{参照要素}\,(\text{更新要素})[a] \div \text{参照要素}\,[b] \end{cases} \tag{4–18}$$

式 (4–18) を利用し，MVA アルゴリズムを以下に示す．

1. 係数行列 A に対応する各要素の演算レベルを格納する行列を準備し，初期値を 0 とする．

2. LU 分解によるシンボリック分解を行う．式 (4–18) より積差演算であれば三つの参照要素のレベルの計算を行う．また，除算であれば二つの参照のレベルの計算を行う．計算されたレベルから最大のものをその演算のレベルとして，手順 1 で準備した配列の該当箇所に設定する．レベル付けのアルゴリズムはそれぞれ以下 (4–19) のように行う．

$$\begin{cases} LEVEL(\text{更新要素}\,[a]) = MAX(LEVEL(\text{参照要素}\,(\text{更新要素})[a]), \\ \qquad\qquad LEVEL(\text{参照要素}\,[b]), LEVEL(\text{参照要素}\,[c])) + 1 \\ LEVEL(\text{更新要素}\,[a]) = MAX(LEVEL(\text{参照要素}\,(\text{更新要素})[a]), \\ \qquad\qquad LEVEL(\text{参照要素}\,[b])) + 1 \end{cases} \tag{4–19}$$

   参照要素となっている b，c はレベルの更新を行わない．

3. 抽出されたレベルの低い順から実行を行う．このとき，同一レベルの演算には互いに依存関係がないため，同時に実行が可能となる．

以上のようなアルゴリズムによって，ベクトル化抽出を行う手法が MVA である．

4.3.2　拡張レベル付けベクトル化 LU 分解法

LU 分解法にかかるそれぞれの計算量は，密行列の行列分解に $O(n^3)$ であり，前進後退代入に $O(n^2)$ である．大規模行列では行列分解に全体の計算量の大部分を占めることから，前進後退代入の計算量の割合は，行列分解にかかる計算量からみると遥かに小さい．

しかし，電子回路シミュレーションや電力潮流計算などのランダムスパース係数行列を扱う分野では，ゼロ要素の割合が 90%以上，各行に非ゼロ要素が数個という極めて高いスパース性を持つ方程式を求解する [31]．このため，密行列のようにはならない．LU 分解，前進代入，後退代入の計算量は，約 7 : 3 : 2 であることがわかっている [31]．しかし，高いスパース行列を扱う場合，LU 分解部分の計算量が減ることで，前進後退代入の計算量の割合が約 40% と大きくなることがわかっている [31]．このため，MVA による行列分解部分のみのベクトル化抽出手法では十分なベクトル化抽出が行えていないことが言える．

LU 分解法は，二つの演算によって計算可能である．このため，LU 分解，前進代入，後退代入の LU 分解法全体で MVA を適用することが可能である．そこで，拡張 MVA は前進後退代入を含めた LU 分解法の処理全体からベクトル化抽出を行うことで，MVA よりもベクトル化効率をあげる．拡張 MVA の手順は，MVA と同様にして前進後退代入にも MVA と同じ手順で，行列分解のレベル付けが終わり次第，前進後退代入部にも MVA を行う．

このとき，前進代入の演算を行う際に，後退代入の演算が追い越さないように注意する必要がある．前進後退代入において，通常は第一行，第二行と行ごとに計算を行うが，拡張 MVA では，第一列，第二列と列ごとに計算を行う．各要素を更新する際に，i 列目の計算が全て終了した時点での最大の演算レベルを更新要素に設定を行うことで，演算の追越を防ぐ．

このように，LU 分解法全体へ MVA を適用することで，それぞれの過程全体から並列性を抽出することで，高いベクトル長を維持することが可能となる．

4.4　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，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と，分岐処理によるワープダイバージェンスが発生するため実行効率が低下する．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が，GPU の構造上そのコードで直接実行することはできない．マシンコードを直接実行できない環境において，演算の種類と演算に必要な要素が格納されている配列のインデックスから構成される情報を配列に格納し，それを解釈実行する手法が知られている [39]．これらを踏まえて，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，配列に格納する．

GPU 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，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して同時に実行する命令を抽出する．図 4–5 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 4–5 中のシンボリック分解では，LU 分解法を一通りシミュレートし，式 (4–18) に示す演算に対応する命令データを，図 4–6 に示すような情報で構成して生成し，配列に先頭から順に格納する．除算演算の命令データは，演算の識別子と演算を行うために 2 要素のアドレスから構成される．積差演算の命令データは，演算の識別子と演算を行うために 3 要素のアドレスから構成される．配列に格納した命令データを 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．カウントされた値は，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タを生成する．生成したベクトルデータは，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スの発生が減少するため，求解を高速化できると考えら

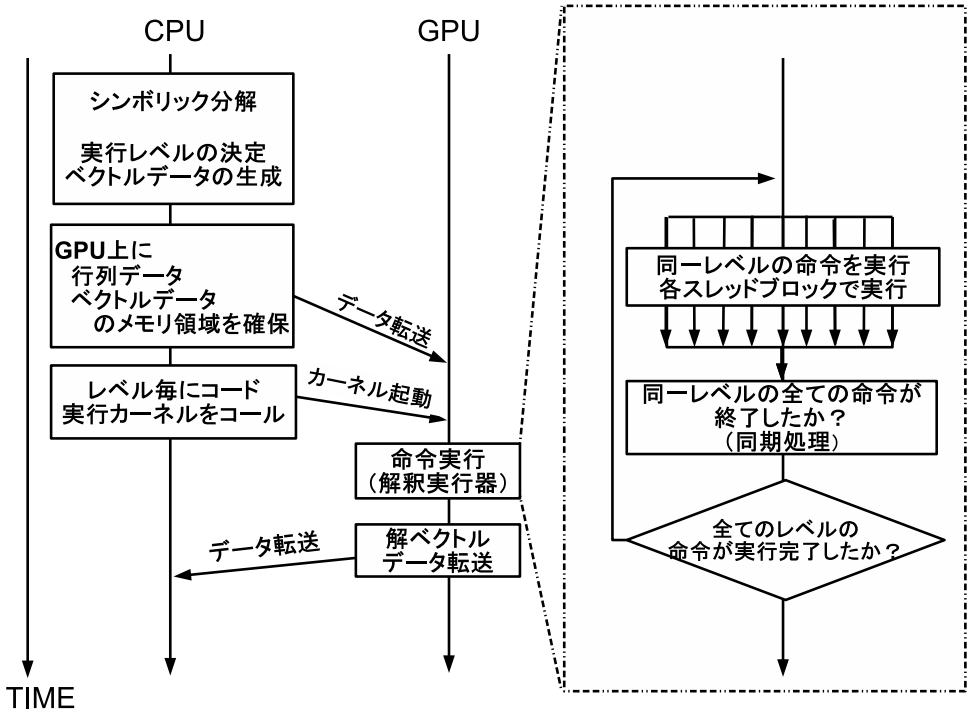

図 4–5 ： 提案手法の実行手順

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 4–6 ： 命令データ

れる．以下では，ワープダイバージェンスを回避する命令の抽出方法と，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．

4.4.1　ワープダイバージェンス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 4–7 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 4–7 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り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 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．

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．

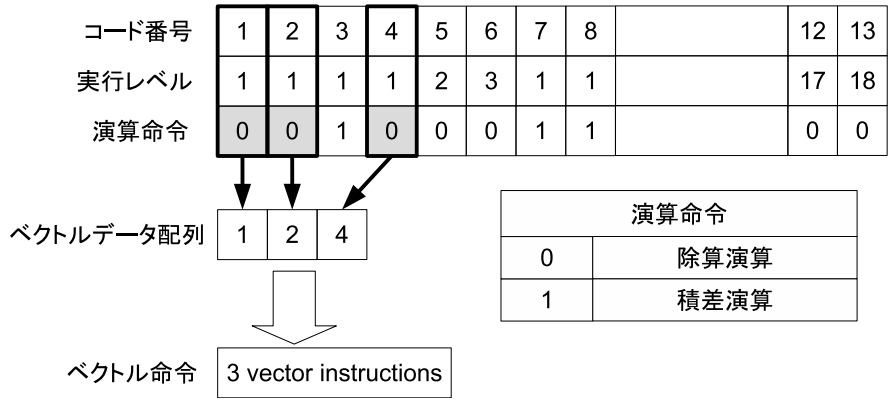

図 4–7 ： 同時レベル付けによる同一演算のベクトル化

4.4.2　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 4–8 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

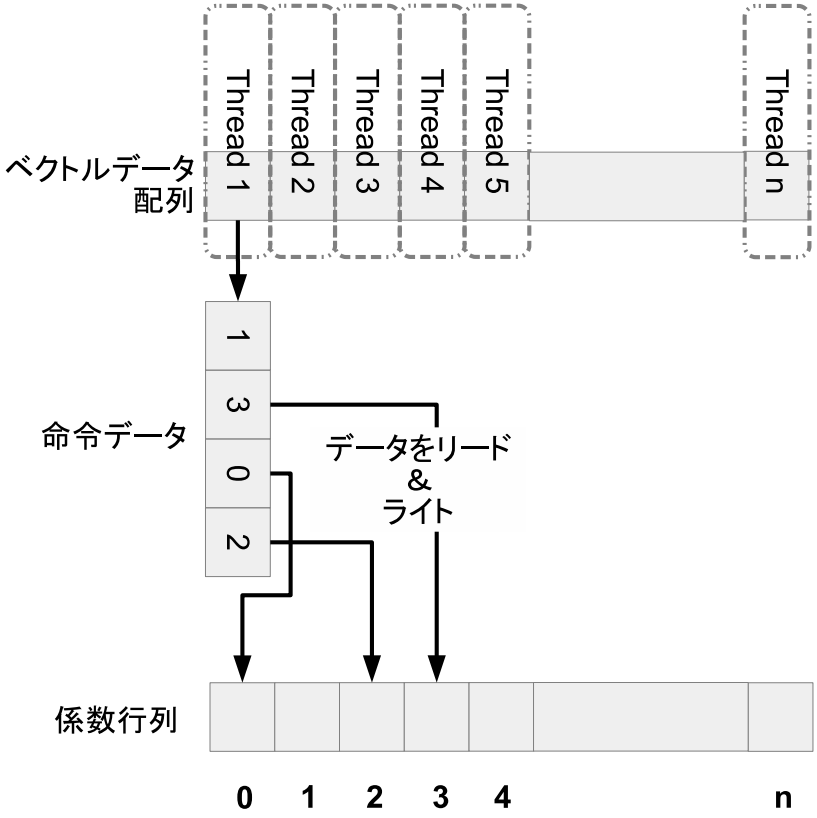

図 4–8 ： ベクトルデータの解釈実行器カーネル

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. 全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）から手順（4）を繰り返す．

表 4–1 ： 評価環境

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

4.5　評価

GPU のワープダイバージェンス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[40]，Matrix Market[41] 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 4–1 に評価環境，表 4–2 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 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 4–2 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．

表 4–2 ： 評価問題

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

表 4–3 ： 拡張ベクトル化 LU 分解法と提案手法のベクトル長

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

4.5.1　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ス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 4–3 に示す．表 4–3 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

表 4–4 ： 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

Name	拡張ベクトル化 LU 分解法		提案手法		高速化率
	実行時間 [ms]	分岐回数 平均：最大	実行時間 [ms]	分岐回数 平均：最大	[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

要なベクトルデータの中で最大のベクトル長である． 表 4–3 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ス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 4–4 に示す．表 4–4 中の分岐回数は，NVIDA Visual Profiler（nvvp）[30] を用いて図 4–5 に示すようにレベル毎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 4–4 中の分岐回数の平均値は式（4–20），最大値は式（4–21）により求める．

$$\text{分岐回数の平均} = \frac{\sum_i \text{ベクトルデータ } i \text{ の分岐回数}}{\text{ベクトルデータの発行回数}} \tag{4–20}$$

$$\text{最大分岐回数} = \max_i \text{ベクトルデータ } i \text{ の分岐回数} \tag{4–21}$$

表 4–4 より，提案手法は，拡張ベクトル化 LU 分解法に対して，全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スの発

表 4–5 ： Warp の実行効率

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 4–6 ： 命令の割合

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

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ス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 4–5 に nvvp で計測したワープの実行効率を示す．表 4–5 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 は全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．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 4–6 に示す．表 4–6 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わかる．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．

4.5.2　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一つである SuperLU_MT[42] を表 4–1 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 4–5 に示すシンボリック分

表 4–7 ： 提案手法と SuperLU_MT の LU 分解法の実行時間

Group	Name	SuperLU_MT	提案手法 [ms]		高速化率
		実行時間 [ms]	実行時間	スケジューリング時間	[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	–

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 上に転送し，ベクトルデータを 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 4–7 に示す．表 4–7 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，提案手法は全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．本章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．

表 4–8 ： CULA ルーチンと提案手法による実行時間

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

4.5.3　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一つである CULA[43] を用いる．提案手法と CULA ソルバの実行時間を表 4–8 に示す．表 4–8 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，全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．

4.6　本章のまとめ

本章では，GPU を用いて実非対象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ス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．

第5章

並列度に応じたハイブリッド並列化手法による高速化

5.1 はじめに

本章では，CPU と GPU で物理メモリを共有するアーキテクチャを利用し，並列度に応じて CPU と GPU を使い分けることで計算を高速化する手法を提案する．

GPU は，CPU とバスで接続されており，ホストメモリとのバスを介したデータ転送が必須となる．また，GPU は，SIMT アーキテクチャであるため，並列性の高い演算には高い性能を得られるが，並列性の低い演算にはカーネルの起動オーバーヘッドが大きくなる．

従来研究では，GPU を用いた多くの数値解析の計算において GPU だけでなく CPU にも計算処理を割当てることで数値解析をより高速化させる手法が提案されている [20][44]．これらの手法は，多くの並列化可能なデータを GPU に割り当て計算をする間に，アイドル状態の CPU も利用することで問題全体の計算を高速化する．一方で，並列化の難しい拡張ベクトル化 LU 分解法は，高い並列性を得ることができるが，問題の行列形状によっては十分な並列性が得られないことがある．このため，複数のスレッドブロックを生成できないような並列性しか得られない場合は，GPU の実行効率が低下する．従来研究では，十分高い並列性を持つデータに対するハイブリッド方式の研究は行われているが，並列性に応じた研究は，GPU では行われていない．一方で，CPU においては，ランダムスパー

ス方程式の求解において並列性によって CPU と GPU を切り替える手法が提案されている [34]．本手法は，LU 分解時に並列性が低い場合は，パイプライン方式の実行方式に切り替えて計算することで並列化のオーバーヘッドを軽減させて計算を高速化する．本手法を GPU で適用する場合，CPU と GPU で並列度に応じて CPU/GPU のカーネルを切り替えることで実現できる．しかし，CPU と GPU はバスを介したデータ転送が必要となるため，並列度に応じたカーネルの切り替えを行うと CPU と GPU 間でデータ転送のコストが増大する．

この問題を解決できるアーキテクチャに，TegraX1 がある．TegraX1 は，CPU と GPU が 1 チップに実装されたヘテロジニアスプロセッサである．本プロセッサの CPU と GPU のメモリアクセスは，共有の DRAM メモリにアクセスする．このとき，CPU と GPU のメモリはそれぞれ確保する必要はあるが，データのコピーは同一の物理メモリ内で行われるためコピー時間が短い．このため，TegraX1 は，PCI-Express などのバスに接続されたノーマルなタイプの GPU である Tesla や Titan などに比べて CPU と GPU でカーネルの切り替えのオーバーヘッドを小さくすることができる．CPU と GPU でカーネルを切り替えて拡張ベクトル化 LU 分解法を実行する場合は，係数行列のデータを切り替えるごとに転送する必要があるが，TegraX1 を使うことでバスを介さないデータ転送が可能となり，計算を高速化できると考えられる．

本章では，TegraX1 を使った拡張ベクトル化 LU 分解法を実装するために，以下のような章構成で述べる．まず，TegraX1 のアーキテクチャについて述べ，次に，4 章で述べた CUDA を用いた拡張ベクトル化 LU 分解法に CPU と GPU のスレッドを切り替えるために追加する切替アルゴリズムについて述べ，最後に本手法を評価する．

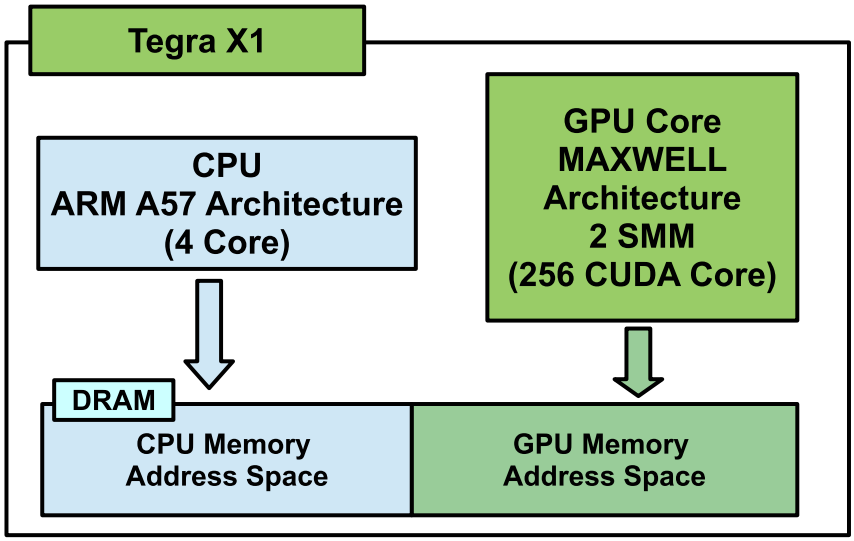

図 5–1 ： ベクトルデータの解釈実行器カーネル

5.2　Tegra X1

TegraX1 は，CPU と GPU を一つのチップに実装したヘテロジニアスプロセッサである．図 5–1 に TegraX1 のアーキテクチャを示す．CPU は ARM Cortex A57 プロセッサの 4 コア，GPU は NVIDIA Maxwell アーキテクチャの 2 SMM（256 CUDA Core） を搭載する．メモリは，DRAM 4GB である．本メモリは，図に示すように CPU と GPU で共有する．TegraX1 は．従来の GPU を搭載するシステムと異なり，CPU と GPU のメモリが物理的に分かれていないため，CPU と GPU で頻繁にスレッドを切り替える場合もメモリアクセスのオーバーヘッドを小さくできる．メモリアクセスには，Unified-Memory の機能を用いる．本機能は，Kepler アーキテクチャから追加された機能であり，CPU と GPU のメモリを統一的に扱い，同一のポインタで扱えるようにした機能である．

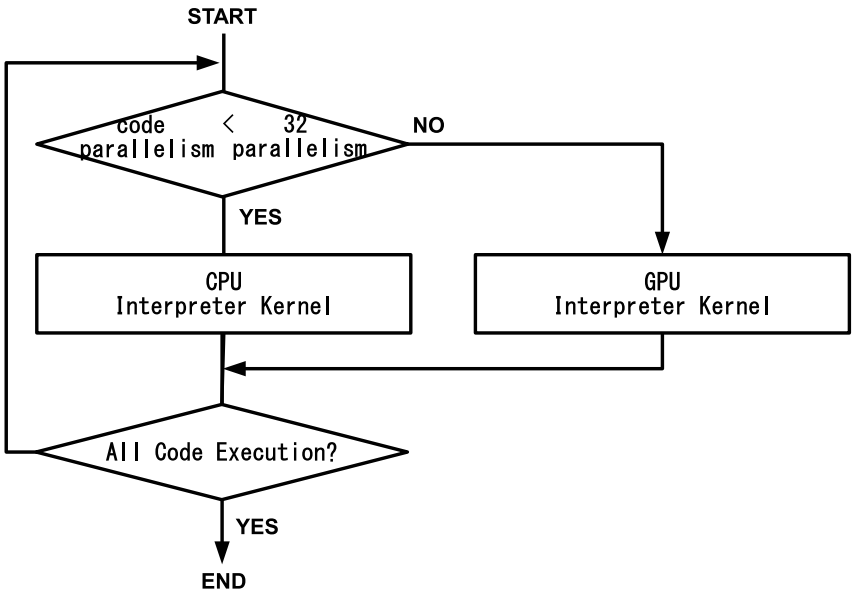

図 5–2 ： CPU/GPU カーネルの切り替えアルゴリズム

5.3　CPU と GPU カーネルの切替手法

拡張ベクトル化 LU 分解法により抽出された命令レベルにおいてベクトル長が短いものは，GPU で実行すると効率が低下する．そこで，本提案では CPU と GPU カーネルを切り替えるためのインタプリタ関数を作成する．切り替えの判定には，拡張ベクトル化 LU 分解法で抽出した各実行レベルのベクトル長のデータを用いて，設定した閾値による判定を行う．図 5–2 に，CPU と GPU を切り替えるアルゴリズムを示す．本例は，閾値を 32 とした場合の例であり，図 5–2 に示す切替アルゴリズムは，CPU を用いて行う．まず，各実行レベルの命令を実行する際に，各実行レベルの命令のベクトル長をベクトル長配列を用いて確認する．次に，確認したベクトル長が閾値（32）以下の場合は，CPU 上でインタプリタカーネルを実行し，ベクトル長が閾値（32）以上の場合は，GPU 上でインタプリタカーネルを実行する．図 5–3 に，切替カーネルの疑似コードを示す．従来手法では，CPU と GPU 上で異なるメモリ空間を確保して実行を行うが，提案手法では同一のメモリ

```
void switch_interpreter(){

   // code array Alloc
   cudaMallocManaged(code);
   // matrix array Alloc
   cudaMallocManaged(matrix);

   // device pointer of host pointer setting
   CudaHostGetDevicePointer(&code);
   // device pointer of host pointer setting
   cudaHostGetDevicePointer(&matrix);


   //instruction-Level execution loop
   for( i < 0; i < MAX_LEVEL; i++){

      //checking parallelism
      int parallelism = level[i];

      //kernel switching
      If( parallelism < 32){

         //Call CPU interpreter kernel
         cpu_interpreter();
      }
      else{

         //Call GPU interpreter kernel
         gpu_interpreter<<<,>>>();

         cudaDeviceSynchronize();
      }
   }
}
```

図 5–3 ： CPU/GPU 切り替えカーネルの疑似コード

空間を用いるために Unified-Memory の機能である Managed 関数を使い CPU と GPU でポインタを共有化する．以降のベクトル長ごとに CPU と GPU を切り替える場合も，同一のポインタを設定し，実行する．

表 5–1 ： JetsonTX1

GPU	NVIDIA Maxwell，256 CUDA Cores
Memory	4GB
CPU	Quad ARM A57 Cores
CUDA Version	9.0

表 5–2 ： 評価問題

Group	Name	行列サイズ
Random	Random	8,000
Bai	dwa512	512
Hamm	add32	4,960
Hamm	memplus	17,758
Bomhof	circuit_3	12,127
Rajat	rajat09	24,482

5.4 評価

TegraX1 を用いた拡張ベクトル化 LU 分解法の高速化手法の有効性を評価するために，TegraX1 を搭載する開発ボードである JetsonTX1 を用いる．拡張ベクトル化 LU 分解法の計算を全て GPU で行う手法を従来手法，CPU と GPU を切り替えて実行する拡張ベクトル化 LU 分解法を提案手法としてランダムスパース方程式を求解し．実行時間を評価する．

評価には，Florida Sparse Matrix Collection[40]，Matrix Market[41] 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 5–1 に JetsonTX1 の環境，表 5–2 に求解する問題を示す．本評価における数値精度は，単精度である．表 5–2 中のグループと名前は，問題のグループ名と問題名である． 表 5–2 の実行結

果を表 5–3 に示す．

表 5–3 ： 実行時間の評価

問題名	従来手法の実行時間 (s)	提案手法の実行時間 (s)	高速化率 (%)
Random 8000	0.0129	0.0081	1.59
dwa512	0.1500	0.1401	1.07
add32	0.0183	0.0148	1.24
circuit_3	0.459	0.248	1.85
memplus	3.497	0.132	26.49
rajat09	0.006	0.005	1.03

評価の結果，memplus のときに最大約 26 倍の高速化が確認できる．これは，ベクトル長の短い実行レベルの時に GPU ではなく CPU により命令を処理したことで高速化できたと考えられる．そこで，ベクトル長が閾値 32 以下に満たない実行レベルの影響を調べるために，実行レベルの数を測定する．図 5–5 から図 5–9 に各問題の実行レベルごとのベクトル長のグラフを示し，表 5–4 に閾値 32 以下のベクトル長の数と実行時間を示す．

図 5–5 から図 5–9 の横軸は実行レベル，縦軸はベクトル長，ベクトル長の 10 と 100 の間にある線は閾値 32 の線を表す．表 5–4 の実行レベルの数はベクトル長が 32 以下の実行レベルの数，実行命令数はベクトル長が 32 以下の実行命令数の総命令数，実行時間はベクトル長が 32 以下の実行レベルを実行するのにかかった時間である．図 5–5 から図 5–9 より，全ての問題において 32 以下のベクトル長となる実行レベルの命令が 32 以上の実行レベルの命令と交互に入れ替わっていることが確認できる．このような場合は，CPU と GPU のスレッドが切り替わり，データ転送が頻繁に発生していると考えられ，処理時間に影響を与えていると考えられる．表 5–3 より，最も高速化できた memplus と最も高速化率が得られなかった rajat09 に着目する．まず，表 5–4 より，memplus におけるベクト

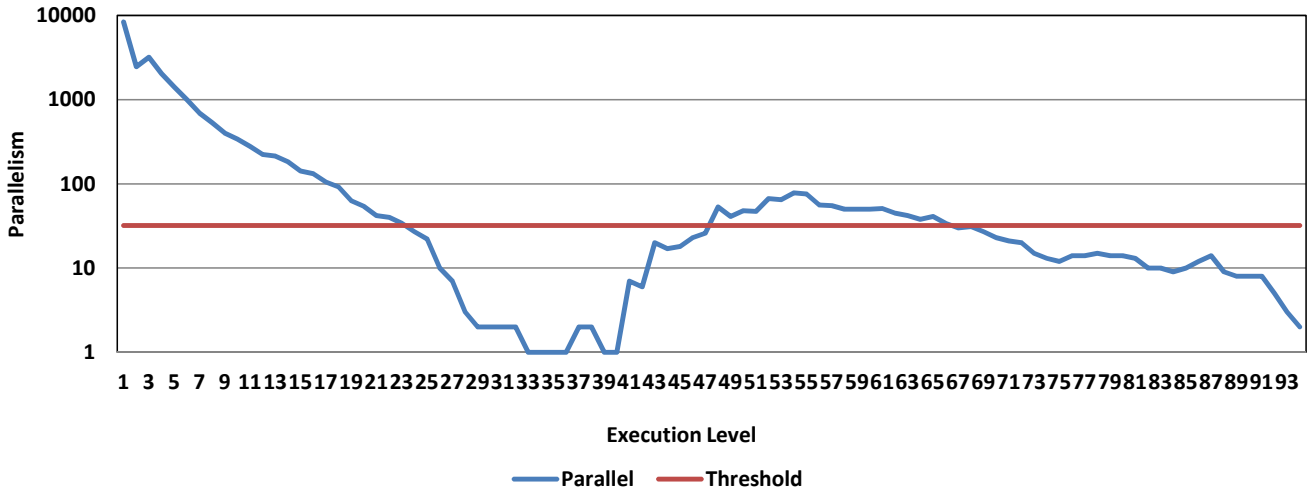

図 5–4 ： 8000 における実行レベルごとのベクトル長

ル長が閾値 32 以下の実行命令数は 2201 であり，求解に必要な実行レベルの約 68%が短いことがわかる．本問題の実行レベルごとのベクトル長の推移図 5–8 より，前半の実行レベル以外の大分部の実行レベルは，並列度が低いことがわかる．このため，並列度の低い実行レベルを CPU で処理したことにより，約 68%のカーネル起動コストを削減できたため高速化できた．次に，rajat09 におけるベクトル長が閾値 32 以下の実行命令数は 26944 であり，求解に必要な実行レベルの約 12.5%が短いことがわかる．本問題の実行レベルごとのベクトル長の推移図 5–9 より，CPU で処理した実行レベルは 4 ととても少ない．このため，多くの実行レベルが GPU 上で効率的に実行できたために，ハイブリッド手法による効果があまりでなかったと考えられる．以上のことから，本手法は，並列度の低い実行レベルを CPU を使って計算することで高い演算性能を得られることがわかる．

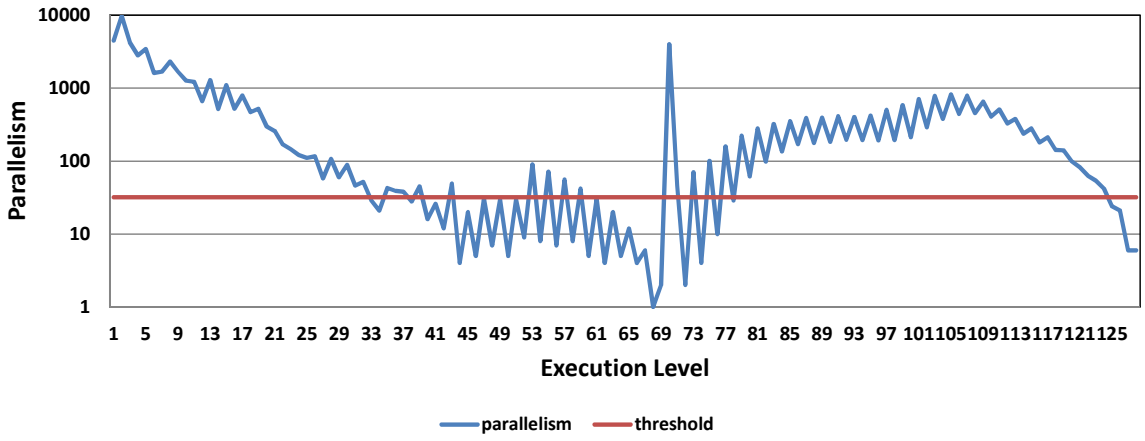

図 5–5 ： add32 における実行レベルごとのベクトル長

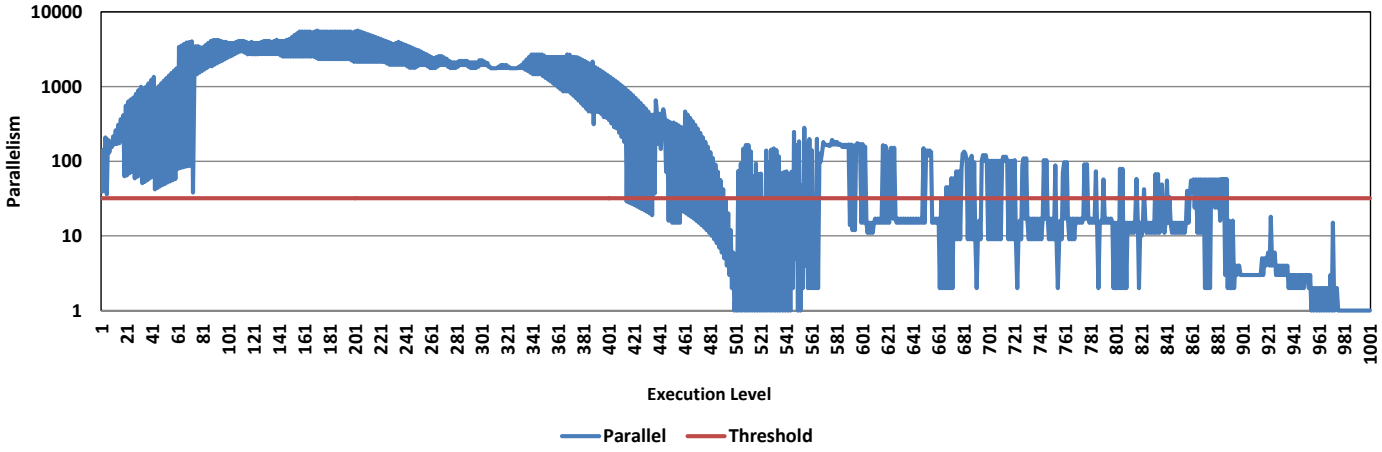

図 5–6 ： dw512 における実行レベルごとのベクトル長

5.5　本章のまとめ

本章では，GPU を用いた拡張ベクトル化 LU 分解法を高速化するために，TegraX1 を用いて CPU と GPU カーネルをベクトル長の閾値を設定し切り替えて実行する手法を提案した．本提案手法を，ヘテロジニアスなプロセッサである TegraX1 に適用することで，バスを介さずに GPU で計算したデータと CPU で計算したデータを転送できる．このた

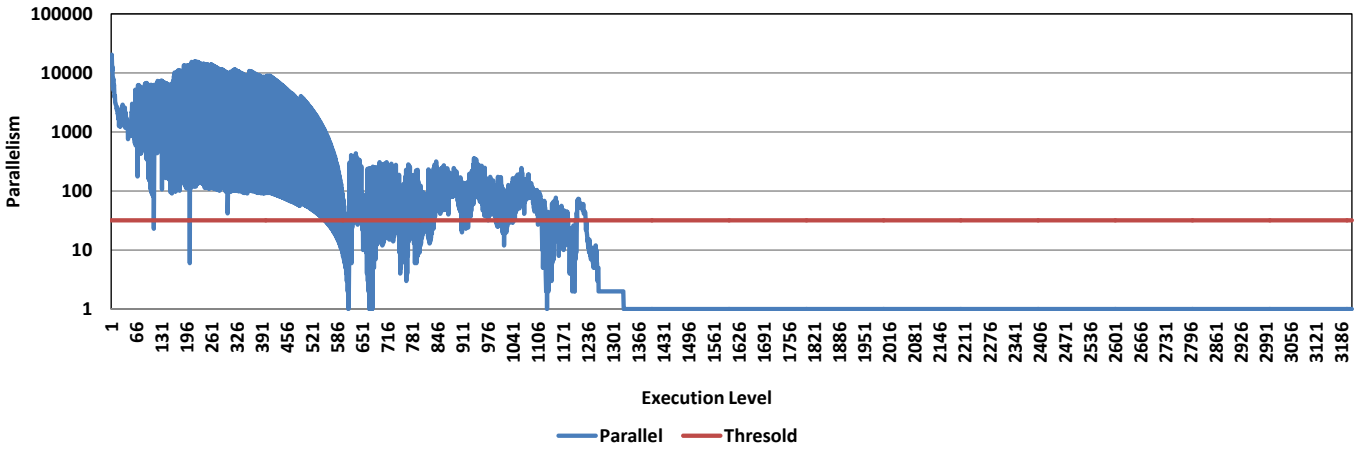

図 5–7 ： circuit_3 における実行レベルごとのベクトル長

め，ベクトル長が長い場合は GPU を用いて，低い場合は CPU を用いるようにカーネルを切り替えてもデータアクセスのコストは，少ないコストで済む．提案手法は，全ての実行レベルを GPU で実行する従来の手法に比べて最大約 26 倍の高速化が確認できた．

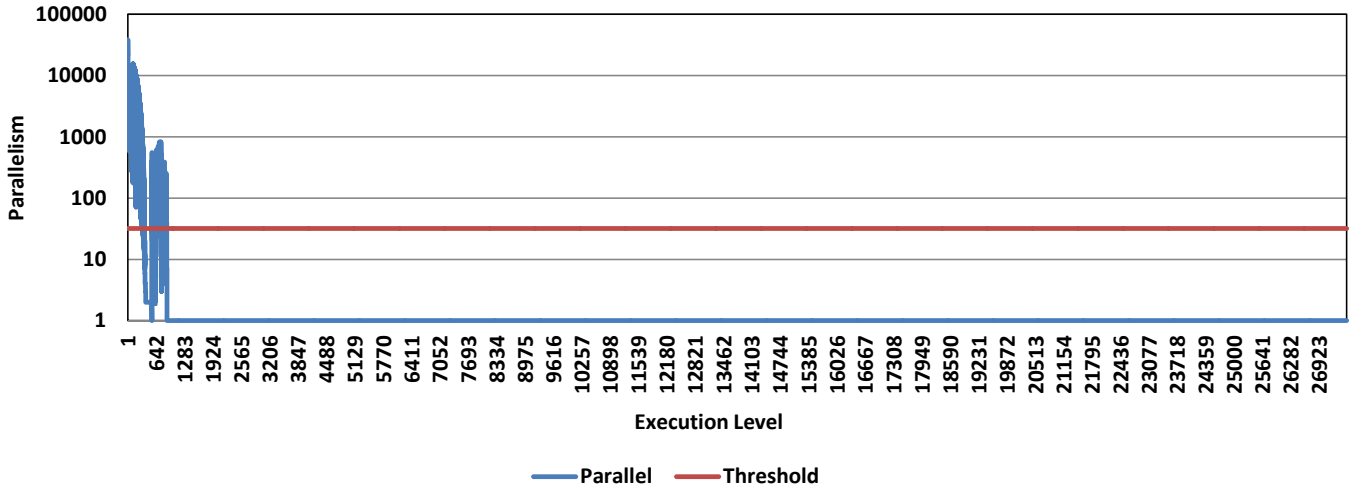

図 5–8 ： memplus における実行レベルごとのベクトル長

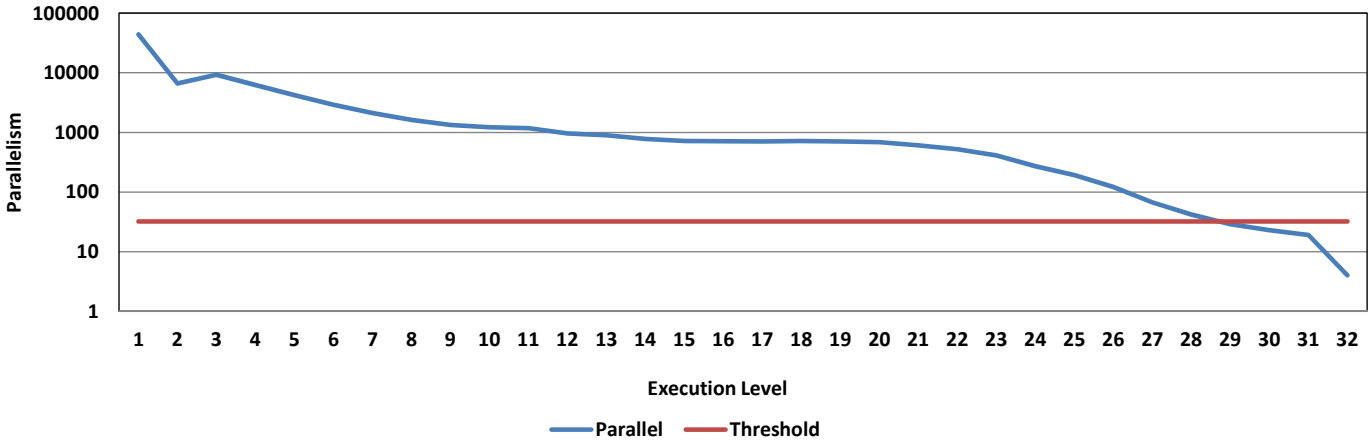

図 5–9 ： rajat09 における実行レベルごとのベクトル長

表 5–4 ： 閾値 32 以下のベクトル長の命令数とその実行時間

問題名	実行レベルの数	実行命令数	**CPU** による 実行時間 (s)	**GPU** による 実行時間 (s)	高速化率 (%)
8000	52	588	0.00001	0.0071	700.6
dw512	397	3780	0.000074	0.0547	740.0
add32	36	486	0.00002	0.00500	250.0
circuit_3	2201	5622	0.00014	0.273	1956.27
memplus	26944	28461	0.00038	3.295	8673.3
rajat09	4	75	0.0001	0.000946	94.60

第6章

結論

本章では，これまでに述べてきた提案手法とその評価結果をまとめ，本研究全体の総括を行う．

本研究では，GPU を用いた数値シミュレーションを高速化するために，GPU のアーキテクチャ構成を活かして実行効率を向上する手法を提案し，その有効性を評価した．

第 3 章「メモリアクセスの局所性の向上」では，格子ボルツマン法を用いたポアズイユの解析においてメモリ階層を効率的に利用するために，階層的にテンポラルブロッキングを行う手法を提案した．GPU は，各スレッドの計算に必要なデータをレジスタ上に保持することで，時間的・空間的な局所性を確保することが可能である．このため，本手法は，シェアードメモリの局所性を向上するためにテンポラルブロッキングに加えてレジスタの局所性を向上させるためのテンポラルブロッキングを行い，GPU 内のメモリアクセスコストを削減する．評価の結果，提案手法は，提案手法を用いない手法と比較して，最大約 7.36 倍高速化できることを確認した．

第 4 章「並列性の抽出による高速化」では，ランダムスパース方程式求解において GPU の実行効率を高めるために，SIMT の実行形式を考慮した並列性の抽出手法を提案した．GPU は，ワープ単位で実行することで効率的に実行されるため，ワープ内が全て同一の演算で実行できるようにすることで計算を高速化できる．本手法は，ランダムスパース方程式求解において，ワープ内のスレッドに同じ演算が割当てられるように依存関係のない命令をベクトル化する．これにより，ワープダイバージェンスを削減する．評価の結果，

提案手法は，数値計算ライブラリのひとつである CULA の LU 分解ルーチンを利用した場合に比べ，最大約 238 倍高速化できることを確認した．また，提案手法は，GPU のアーキテクチャに合わせた最適化を行わない手法に比べ，最大約 1.6 倍高速化できることを確認した．

第 5 章「並列度に応じたハイブリッド並列化手法による高速化」では，ランダムスパース方程式求解において，依存関係を持つ演算をハイブリッドに並列化する手法を提案した．本手法は，拡張ベクトル化 LU 分解法によるランダムスパース方程式求解において，各実行レベルの演算をベクトル長の閾値によって CPU と GPU のどちらに割り当てるかを決定する．CPU と GPU でバスを介したデータ転送が不要なアーキテクチャである TegraX1 を用いて，評価を行った．評価の結果，提案手法は，実行を切り替えない手法に比べ，最大約 26 倍高速化できることを確認した．

以上の結果より，複数の数値解析計算を GPU のアーキテクチャに適した最適化を行い，その効果を確認した．第 3 章で提案した手法は，一般的なステンシル計算を用いた解析アプリケーション全般に適用可能であり，第 4 章や第 5 章で提案した手法は，動的に並列性を抽出するアプリケーションに適用可能である．特に第 5 章で提案した手法は，通信のオーバーヘッドに対して閾値を設定することで解析処理を高速化できる．

また，第 4 章や第 5 章で述べた連立一次方程式求解におけるベクトル化とハイブリッド手法は，3 章のような LBM を含めた様々なステンシル計算において方程式を求解するようなアプリーケーションにも適用することができる．この場合，各ブロックの計算から同一演算が可能な命令を抽出し，ベクトル長に応じて切替手法を適用する．このような実装では，ブロック内のスレッドに割り当てられた計算の並列性を各格子点座標を用いずに命令レベルで抽出できる．ただし，格子点データだけでなく命令レベルの情報を保持する必要が生じるため，使用メモリ量が多くなる．このため，高い高速化率を得るためにはテンポラルブロッキングの段数や CPU-GPU で計算を切り替える閾値であるパラメータなど

を適切に設定する必要がある.

謝辞

本研究は，千葉工業大学大学院 情報科学研究科 情報科学専攻の在学期間中になされたものです．非常に多くの方々の協力によって本研究を博士論文としてまとめることができました．ここに謝意を表したいと思います．

研究室配属からの長い期間にわたって御指導ご鞭撻を賜りました指導教員の前川仁孝教授，宮崎収兄 教授には，多くの助言を頂きましたことを深く感謝を致します．

富井規雄 教授，藤田茂 教授をはじめとする本学 情報工学科の先生方には，研究面だけでなく多くの面で支えて頂きました．本当にありがとうございました．

また，本博士論文をまとめるにあたり，多くの助言を頂きました早稲田大学 木村啓二教授には，深く感謝を致します．

研究面や生活面でお世話になりました先輩方や同輩，後輩に感謝致します．

そして最後に，両親，友人，および本論文を執筆できる環境へおいてくださった全ての皆様に感謝致します．

2018 年 10 月 27 日

参考文献

(1) 越村俊一，香月恒介，茂渡悠介：GPU コンピューティングによる津波解析の高速化とリアルタイム浸水予測，土木学会論文集 B2(海岸工学)，Vol. 66, No. 1, pp. 191–195 (2010).

(2) Shimokawabe, T., Aoki, T. and Onodera, N.: High-Productivity Framework on GPU-Rich Supercomputers for Operational Weather Prediction Code ASUCA, *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 251–261 (2014).

(3) Fujita, K., Yamaguchi, T., Ichimura, T., Hori, M. and Maddegedara, L.: Acceleration of Element-by-Element Kernel in Unstructured Implicit Low-Order Finite-Element Earthquake Simulation Using OpenACC on Pascal GPUs, *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*, pp. 1–12 (2016).

(4) 及川貴瑛，園田　潤，佐藤源之，本間規泰，池川豊年：3 次元 MW-FDTD 並列計算を用いた大規模地形モデルにおける雷放電電磁界解析，電気学会論文誌 A（基礎・材料・共通部門誌），Vol. 132, No. 1, pp. 44–50 (2012).

(5) Monaghan, J. and Kocharyan, A.: SPH simulation of multi-phase flow, *Computer Physics Communications*, Vol. 87, No. 1, pp. 225 – 235 (1995). Particle Simulation Methods.

(6) Krawezik, G. P. and Poole, G.: ccelerating the ANSYS direct sparse solver with GPUs, 2010 Symposium on Application Accelerators in High Performance Comput-

ing (SAAHPC ' 10) (2010).

(7) Quarles, T.: The SPICE3 Implementation Guide, Technical Report UCB/ERL M89/44, EECS Department, University of California, Berkeley (1989).

(8) Adams, S., Payne, J. and Boppana, R.: Finite Difference Time Domain (FDTD) Simulations Using Graphics Processors, *2007 DoD High Performance Computing Modernization Program Users Group Conference*, pp. 334–338 (2007).

(9) Gan, J., Zhou, Z. and Yu, A.: A GPU-based DEM approach for modelling of particulate systems, *Powder Technology*, Vol. 301, pp. 1172 – 1182 (2016).

(10) Feng, Z. and Li, P.: Multigrid on GPU: Tackling Power Grid Analysis on Parallel SIMT Platforms, *Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, ICCAD '08, pp. 647–654 (2008).

(11) Godel, N., Nunn, N., Warburton, T. and Clemens, M.: Scalability of Higher-Order Discontinuous Galerkin FEM Computations for Solving Electromagnetic Wave Propagation Problems on GPU Clusters, *IEEE Transactions on Magnetics*, Vol. 46, No. 8, pp. 3469–3472 (2010).

(12) NVIDIA TESLA V100 GPU ARCHITECTURE, available from<http://images. nvidia .com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf> (accessed 2018-10-15).

(13) Radeon's next-generation Vega architecture, available from<https://radeon.com/ _downloads/vega-whitepaper-11.6.17.pdf> (accessed 2018-10-15).

(14) Lindholm, E., Nickolls, J., Oberman, S. and Montrym, J.: NVIDIA Tesla: A Unified Graphics and Computing Architecture, *IEEE Micro*, Vol. 28, No. 2, pp. 39–55 (2008).

(15) NVIDIA CUDA Toolkit, available from <https://developer.nvidia.com/cuda-toolkit/> (accessed 2019/01/30).

(16) KHRONOS GROUP OpenCL, available from <https://www.khronos.org/opencl/> (accessed 2019/01/30).

(17) OpenACC, available from <https://www.openacc.org/> (accessed 2019/01/30).

(18) Maruyama, N. and Aoki, T.: Optimizing Stencil Computations for NVIDIA Kepler GPUs, *1st International Workshop on High-Performance Stencil Computations 2014* (2013).

(19) Calore, E., Marchi, D., Schifano, S. F. and Tripiccione, R.: Optimizing communications in multi-GPU Lattice Boltzmann simulations, *2015 International Conference on High Performance Computing Simulation (HPCS)*, pp. 55–62 (2015).

(20) 大島聡史，吉瀬謙二，片桐孝洋，弓場敏嗣：CPU と GPU を用いた並列 GEMM 演算の提案と実装，情報処理学会論文誌コンピューティングシステム（ACS），Vol. 47, No. 12, pp. 317–328 (2006).

(21) Cheng, J., Grossman, M. and McKercher, T.: *Professional CUDA C Programming*, Wiley (2014).

(22) Jin, G., Lin, J. and Endo, T.: Efficient utilization of memory hierarchy to enable the computation on bigger domains for stencil computation in CPU-GPU based systems, *2014 International Conference on High Performance Computing and Applications (ICHPCA)*, pp. 1–6 (2014).

(23) Jin, G., Endo, T. and Matsuoka, S.: A Multi-Level Optimization Method for Stencil Computation on the Domain that is Bigger than Memory Capacity of GPU, *2013*

IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum, pp. 1080–1087 (2013).

(24) 小野寺直幸，青木尊之，下川辺隆史，小林宏充：格子ボルツマン法による 1m 格子を用いた都市部 10km 四方の大規模 LES 気流シミュレーション，ハイパフォーマンスコンピューティングと計算科学シンポジウム論文集，Vol. 2013, pp. 123–131 (2013).

(25) Rinaldi, P., Dari, E., Venere, M. and Clausse, A.: A Lattice-Boltzmann solver for 3D fluid simulation on GPU, *Simulation Modelling Practice and Theory*, Vol. 25, pp. 163 – 171 (2012).

(26) Suksumlarn, J., Suwannik, W. and Maleewong, M.: Lattice Boltzmann method for two-dimensional shallow water equations with CUDA, *2015 2nd International Conference on Advanced Informatics: Concepts, Theory and Applications (ICAICTA)*, pp. 1–5 (2015).

(27) Qian, Y. H., D'Humieres, D. and Lallemand, P.: Lattice BGK Models for Navier-Stokes Equation, *EPL (Europhysics Letters)*, Vol. 17, No. 6, pp. 479–484 (1992).

(28) 清水圭太，伊澤精一郎，福西　祐，熊　鰲魁：格子ボルツマン法を用いた平面ポアズイユ流れの不安定性の解析，東北支部総会・講演会　講演論文集，Vol. 2005.40, pp. 28–29 (2005).

(29) Yin, X. and Zhang, J.: An improved bounce-back scheme for complex boundary conditions in lattice Boltzmann method, *Journal of Computational Physics*, Vol. 231, No. 11, pp. 4295 – 4303 (2012).

(30) NVIDIA Visual Profiler — NVIDIA Developer, available from<https://developer.nvidia.com/nvidia-visual-profiler> (accessed 2019-01-30).

(31) 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).

(32) Christen, M., Schenk, O. and Burkhart, H.: General-Purpose Sparse Matrix Building Blocks using the NVIDIA CUDA Technology Platform (2007).

(33) Jalili-Marandi, V., Zhou, Z. and Dinavahi, V.: Large-Scale Transient Stability Simulation of Electrical Power Systems on Parallel GPUs, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 23, No. 7, pp. 1255–1266 (2012).

(34) 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 Transactions on Circuits and Systems II: Express Briefs*, Vol. 58, No. 10, pp. 702–706 (2011).

(35) 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*, pp. 2530–2533 (2012).

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

(37) 小国　力，村田健郎：行列計算ソフトウェア：WS、スーパーコン、並列計算機，丸善 (1991).

(38) Markowitz, H. M.: The Elimination form of the Inverse and its Application to Linear Programming, *Management Science*, Vol. 3, No. 3, pp. 255–269 (1957).

(39) 和宜根本，宏史佐田，仁孝前川，光宏伊與田，収兄宮崎：命令キャッシュを考慮したコード生成法による方程式求解の高速化手法，技術報告 80(2004-ARC-159) (2004).

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

(41) Matrix Market, available from<http://math.nist.gov/MatrixMarket/> (accessed 2018-10-15).

(42) Demmel, J., Eisenstat, S., Gilbert, J., Li, X. and Liu, J.: A Supernodal Approach to Sparse Partial Pivoting, *SIAM Journal on Matrix Analysis and Applications*, Vol. 20, No. 3, pp. 720–755 (1999).

(43) CULA Tools-GPU accelerated LAPACK, available from<http://www.culatools.com/> (accessed 2018/10/15).

(44) Venkatasubramanian, S., Vuduc, R. W. and none, n.: Tuned and Wildly Asynchronous Stencil Kernels for Hybrid CPU/GPU Systems, *Proceedings of the 23rd International Conference on Supercomputing*, ICS '09, pp. 244–255 (2009).

研究業績

論文

[1] 松瀬弘明，中村あすか，富永浩文，前川仁孝，”タスクスケジューリング問題における探索ノード数削減による PDF/IHS 法の高速化”，情報処理学会論文誌コンピューティングシステム (ACS)，Vol.11，No.2，pp.17-26，(2018 年 8 月 3 日).

[2] 富永浩文，中村あすか，前川仁孝，”レジスタ最適化を用いた CUDA による格子ボルツマン法の高速化手法”，情報処理学会論文誌プログラミング (PRO)，Vol.11, No.2, pp.1-8，(2018 年 6 月 26 日).

[3] 中村あすか，富永浩文，前川仁孝，”タスクスケジューリング問題におけるレディ状態の割当て削減による PDF/IHS の高速化”，情報処理学会論文誌，Vol.58，No.3，pp654-662，(2017 年 3 月 15 日).

[4] 富永浩文，前川仁孝，”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”，情報処理学会論文誌プログラミング (PRO)，Vol.7, No.1, pp.10-17，(2014 年 1 月 22 日).

国際会議

[1] Hirobumi Tominaga，Asuka Nakamura，Yoshitaka Maekawa，”Evaluation of EMVA using the instruction-level parallelism on TegraX1”，2018 Sixth International Sym-

posium on Computing and Networking(CANDAR)，pp.1–4, (2018 年 11 月 27 日).

[2] Asuka Nakamura, Hirobumi Tominaga，Yoshitaka Maekawa，”Evaluation of Hierarchical Pincers Attack Search on Distributed Memory Systems”，2018 Sixth International Symposium on Computing and Networking(CANDAR)，pp.1–4, (2018 年 11 月 27 日).

査読有シンポジウム

[1] 長坂一生，富永浩文，中村あすか，前川仁孝，”CUDA における JDS 形式疎行列ベクトル積に対するダイナミックパラレリズムの評価”，The 2nd Cross-disciplinary workshop on computing Systems, Infrastructures, and programming (xSIG 2018), (2018 年 5 月 29 日).

[2] 中村あすか，富永浩文，前川仁孝，”探索の重複領域削減による階層的挟み撃ち探索の高速化”，先進的計算基盤システムシンポジウム論文集，Vol.2011，pp.348-355，(2011 年 5 月 18 日).

国内学会・研究会

[1] 富永浩文，中村あすか，前川仁孝，”レジスタ最適化を用いた CUDA による格子ボルツマン法の高速化手法”，第 117 回プログラミング研究会発表会，2017-4-(5)，pp.1–8，(2018 年 1 月 15 日).

[2] 松瀬弘明，中村あすか，富永浩文，前川仁孝，”タスクスケジューリング問題における DF/IHS 法の探索ノード数削減”，第 110 回プログラミング研究発表会，2016-2-(4), pp.1–6, (2016 年 8 月 10 日).

[3] 鈴木宏明，富永浩文，中村あすか，前川仁孝，”GPU を用いた格子ボルツマン法のループ展開を利用した同期オーバヘッド削減による高速化”，第 105 回プログラミング研究発表会，2015-2-(1), pp.1–7, (2015 年 8 月 5 日).

[4] 富永浩文，前川仁孝，”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”，第 95 回プログラミング研究発表会，2013-2-(7), pp. 1–8, (2013 年 8 月 2 日).

[5] 篠塚研太，富永浩文，中村あすか，前川仁孝，”時間依存性のない線形素子による定数伝播を用いた電子回路シミュレータ SPICE3 の高速化”，情報処理学会研究報告ハイパフォーマンスコンピューティング (HPC)，Vol. 2011-HPC-132, No. 3, pp. 1–6, (2011 年 11 月 28 日).

国内学会・全国大会

[1] 松瀬弘明，中村あすか，富永浩文，前川仁孝，”タスクスケジューリング問題における DF/IHS 法のハッシュテーブルを用いた探索ノード数削減”，情報処理学会第 78 回全国大会講演論文集第 1 分冊，4H-06, pp. 197–198, (2016 年 3 月 10 日).

[2] 鈴木宏明，富永浩文，佐藤一馬，中村あすか，前川仁孝，”GPU を用いた格子ボルツマン法のループ展開を利用したメモリアクセスの局所性向上による高速化”，情報処理学会第 77 回全国大会第 1 分冊，3J-06, pp. 71–72, (2015 年 3 月 17 日).

[3] 佐藤一馬，富永浩文，中村あすか，前川仁孝，松林祐，若松真治，”有限要素法による電磁場解析の透磁率を用いた演算回数削減手法”，情報処理学会第 75 回全国大会第 1 分冊，1K-5, pp. 93–94, (2013 年 3 月 6 日).

[4] 松井南実，富永浩文，中村あすか，前川仁孝，”GPU のキャッシュヒット率向上による DEM の高速化”，情報処理学会第 74 回全国大会第 1 分冊，4K-8, pp. 215–216, (2012 年 3 月 6 日).

[5] 富永浩文，中村あすか，篠塚研太，前川仁孝，”GPU のための回路方程式求解における命令レベル並列度の評価”，情報科学技術フォーラム講演論文集，Vol. 10, No. 1, pp. 343-344, (2011 年 9 月 9 日).

[6] 中村あすか，富永浩文，前川仁孝，”探索の重複領域を削減した階層的挟み撃ち探索による実行時間最小マルチプロセッサスケジューリング問題の求解”，情報科学技術フォーラム講演論文集，Vol. 10, No. 1, pp. 379-380, (2011 年 9 月 7 日).

[7] 篠塚研太，中村あすか，富永浩文，前川仁孝，”ストリーミング SIMD 拡張命令を用いた電子回路シミュレータ SPICE3 の高速化”，情報科学技術フォーラム講演論文集， Vol. 9, No. 1, pp. 339–340, (2010 年 8 月 20 日).

受賞

[1] ”2014 年度コンピュータサイエンス領域奨励賞”，第 101 回プログラミング研究発表会 (PRO-2014-3)，(2014 年 11 月 10 日).