

# Octree型AMRを導入した格子ボルツマン法のC++テンプレートを用いたカーネル生成によるGPU計算の高速化

長谷川 雄太<sup>1,a)</sup> 青木 尊之<sup>1,b)</sup>

受付日 2015年11月6日, 採録日 2016年2月3日

**概要:** 格子ボルツマン法に Octree-based Adaptive Mesh Refinement (AMR) を導入すると細分化の末端のリーフに割り当てる格子サイズが小さくなるため, GPU 計算の実行性能が大きく低下する. 特に D3Q27 型では, リーフの外殻部分にある格子 (外殻格子) の計算は, 隣接格子点参照に対して多数の条件分岐を含む. 同一の計算を行いながら外殻格子数を削減するために, Octree の 1 段上のノードを利用した冗長計算を行うマザーリーフ法を提案し, 1.96 倍に計算を高速化させることができた. また, 外殻格子計算における条件分岐を排除するために, すべての参照パターンを列挙し, 異なる GPU カーネル関数に分割する方法を提案した. 参照パターンが 702 通りと非常に多く, 手動での記述は困難である. C++テンプレート展開を用いて参照パターンに対応する GPU カーネル関数を自動生成することにより, プログラムの生産性を維持しながら, 外殻格子の計算速度を 2.29 倍に向上させることができた.

**キーワード:** 適合細分化格子法, AMR, Octree, 格子ボルツマン法, D3Q27, GPU カーネル関数, C++テンプレート

## High-performance GPU-kernel Generation Using C++ Template for a Computation for Lattice Boltzmann Method with Octree-based Adaptive Mesh Refinement

YUTA HASEGAWA<sup>1,a)</sup> TAKAYUKI AOKI<sup>1,b)</sup>

Received: November 6, 2015, Accepted: February 3, 2016

**Abstract:** It is a severe problem that D3Q27 Lattice Boltzmann Method (LBM) with Octree-based Adaptive Mesh Refinement (AMR) has a low GPU computational performance due to small-size meshes assigned to the leaves located at the end of the octree structure. In particular, the computations contains a lot of branch divergences at the outer leaf boundary when it refers to the grid on the neighbor leaves. We propose a method which uses the upper node to reduce the number of grids at the outer leaf boundary by executing redundant computations (Mother-Leaf Method), so that we have achieved 1.96 times speed-up for the same computation. In order to exclude the branch divergences in the GPU kernel function on the grid at the outer leaf boundary, we divide it into individual kernels depending on each different access pattern to the neighbor leaf grid. It is too heavy for a handwritten program to make GPU kernel functions corresponding to 702 access patterns. We utilize the C++ Template expansion to generate GPU kernel functions in compiling and have succeeded in 2.29 times speed-up for the computation at the outer leaf boundary. The automatic generation of numerous GPU kernel functions also makes it possible to keep the productivity of the AMR-LBM code.

**Keywords:** Adaptive Mesh Refinement, Octree, Lattice Boltzmann Method, D3Q27, GPU kernel function, C++ Template expansion

<sup>1</sup> 東京工業大学  
Tokyo Institute of Technology, Meguro, Tokyo 152-8550,  
Japan

a) hasegawa@sim.gsic.titech.ac.jp

b) taoki@gsic.titech.ac.jp

### 1. はじめに

規則的に並んだ格子点データがある一定の規則で更新する計算をステンシル計算といい, 特に隣接する  $n$  個の格子

点の値を用いて計算する場合を  $n$  点ステンシル計算と呼ぶ。たとえば、2次元で上下左右4点と斜め4点の隣接格子および自分自身を参照する9点ステンシル計算、3次元で上下左右・奥・手前の隣接点を参照する7点ステンシル計算、7点に加えて斜め方向まですべての隣接点を参照する27点ステンシル計算がある。ステンシル計算は、拡散方程式や流体方程式を有限差分法などで離散化して計算する際に必要となるため、連続体で記述される物理現象のシミュレーションにおいて最も重要かつ基本的な計算カーネルである。

近年、計算機の急速な発展にともない膨大な数の格子点を扱う大規模計算が可能となり、理工学のさまざまな分野において高解像度計算や広域計算が行われている。特にGPU (Graphics Processing Unit) のようなアクセラレータ (演算加速器) は細粒度の並列化が可能なステンシル計算に適しているため、さまざまな大規模物理シミュレーションに用いられている [1], [2], [3]。

ステンシル計算の一種である格子ボルツマン法 (Lattice Boltzmann Method; LBM) は、非圧縮性流体の解法として広く用いられている。従来の非圧縮性流体の解法に比べて格子ボルツマン法は、大規模な連立一次方程式を解く必要がなく、ベタスケールを超える大規模計算に適した手法として研究が進められている [2], [4], [5], [6], [7]。

これまでの大規模GPU計算では、計算領域の全体を一樣な解像度とする等間隔格子が用いられてきた [8], [9]。等間隔格子はステンシル計算のメモリアクセス効率が良く、高い実行性能を得ることができる。しかし、実際の物理現象に対して、計算の全領域で高い解像度が要求されることは稀であり、等間隔格子による計算は無駄な計算を多く含んでいるといえる。任意の計算空間の解像度を変更する手法を導入することで、高精度かつ大規模な計算を行うことができるようになる。従来の有限要素法や有限体積法では、局所的に解像度を高めることのできる三角形や四面体要素の非構造格子が用いられてきたが、メモリアクセスが間接参照であるため不規則となり、将来的な大規模計算には不向きとされている。

等間隔格子の実行性能を維持しつつ、高解像度の計算が必要な部分に局所的に細かい格子を配置する手法として、適合細分化格子法 (Adaptive Mesh Refinement; AMR, 図1) [10], [11] が使われている。AMRにはBlock-structured, Unstructured, Tree-based などさまざまな種類があるが、なかでも大規模計算に適した手法としてOctree型AMRに関する研究が多く行われている [12], [13]。以降ではOctree型AMRのみを取り上げるため、単にAMRと称した場合にはOctree型AMRを指すものとする。

AMRは、木データ構造で表される再帰的な領域分割によって計算領域中に複数の解像度の格子を割り当てる手法である。図2のように、木データ構造の根ノードは計算領

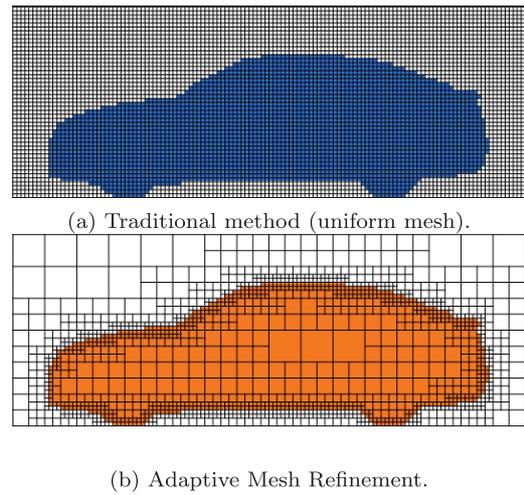
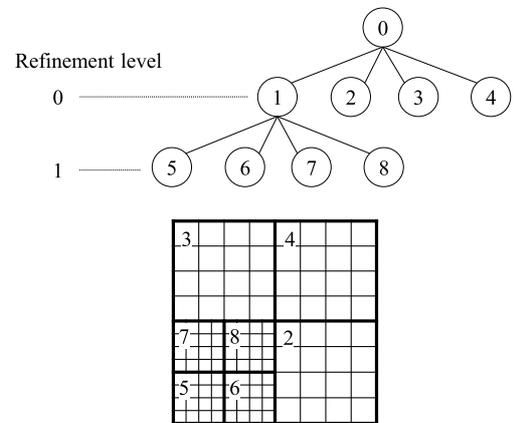
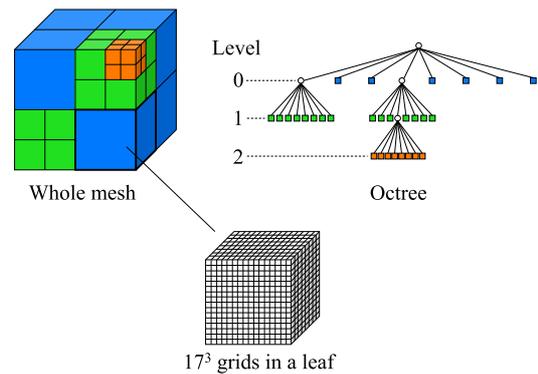


図1 適合細分化格子法 (AMR) の例  
Fig. 1 Example of Adaptive Mesh Refinement (AMR).



(a) 2D mesh (an example explaining the concept).

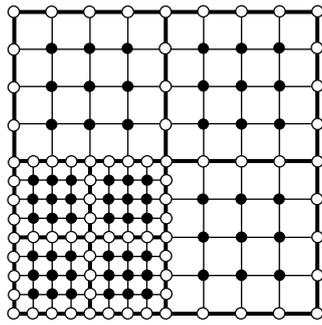


(b) 3D mesh (implemented in this paper).

図2 Octreeに基づくAMRのデータ構造  
Fig. 2 Data structure of octree-based AMR.

域全体に相当し、共通の親を持つ子ノードは親ノードの計算領域を分割した小領域に対応する。葉ノード (リーフ) に対応する小領域には一定の格子点数の格子が割り当てられるため、領域分割の再帰深度が高い場所ほど高解像度の格子が割り当てられる。

AMRの大規模計算に関する既往研究 [14], [15], [16] で



○ Outer shell lattice, ● Inner lattice

図 3 リーフの外殻格子と内部格子

Fig. 3 Outer shell lattices and inner lattices in a leaf mesh.

は、CPU による大規模計算が広く行われているが、GPU 計算では十分な実行性能が得られていない。リーフの格子点数は全計算領域を等間隔格子にする場合と比べて非常に少ないため、GPU で十分な並列数を確保することが難しい。また、ステンシル計算と隣接リーフ間のデータ交換とを分離してプログラム実装するためにリーフに袖領域の格子 (Halo) を設けている場合が多く、計算格子点より多くのメモリ容量が必要となる。GPU は CPU に比べてメモリ容量が小さいため、リーフに袖領域をとる方法では計算格子点数がより厳しく制限される。このため、AMR を導入した大規模なステンシル計算の GPU への実装には課題が多く、高い実行性能を得るためには GPU 計算に適したデータ構造や高速化手法について検討する必要がある。

本論文では、メモリ使用量を抑えるため、リーフに袖領域を設けず隣接するリーフの格子を直接参照する実装方法を採用する。しかし、リーフの境界部分の格子点 (外殻格子, Outer shell lattices, 図 3) でステンシル計算のための隣接点参照のパターンが複雑になり、Naive な実装では実行性能の大幅な低下を招く。この性能低下を改善するため、次の 2 つの手法を導入する。まず、同一の計算を行いながら外殻格子数を削減するためにマザーリーフ法を提案する。Octree のリーフの 1 段上のノードを利用した冗長計算により、総格子点数に占める外殻格子点数の割合を減少させる。次に、複雑なアクセスパターンを適切に分類し、複数の GPU カーネルに分割して計算する方法を試みる。AMR を導入した格子ボルツマン法のアクセスパターンは 702 通りと非常に多数であり、それらを手動で列挙することは現実的ではない。本論文では、C++ テンプレートを用いて GPU カーネル関数を記述し、702 通りのアクセスパターンに対応した GPU カーネル関数を自動生成することで、プログラムの生産性・保守性を維持しながら計算を高速化する。

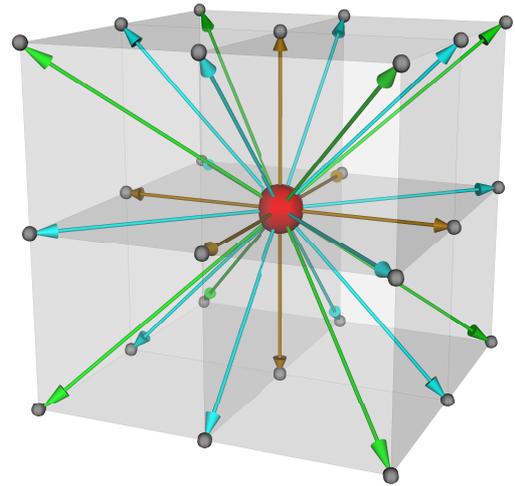


図 4 3次元 27速度 (D3Q27) モデル

Fig. 4 Illustration of D3Q27 velocity model.

## 2. 数値計算手法

### 2.1 格子ボルツマン法

格子ボルツマン法 (Lattice Boltzmann Method; LBM) は、流体をいくつかの離散化速度を持った仮想粒子の集合体と仮定し、仮想粒子の並進と衝突の時間発展を計算する手法である。仮想粒子の速度には、1 タイムステップで粒子がちょうど隣の格子点に移動するような速度が選ばれる。本論文では、仮想粒子の速度として、図 4 に示す 3次元 27速度 (D3Q27) モデルを用いる。D3Q27 モデルにおける粒子の速度は、次のように定義される。

$$c_{\alpha} = \begin{cases} (0, 0, 0), & \alpha = 0 \\ (\pm c, 0, 0), (0, \pm c, 0), (0, 0, \pm c), & 1 \leq \alpha \leq 6 \\ (\pm c, \pm c, 0), (\pm c, 0, \pm c), (0, \pm c, \pm c), & 7 \leq \alpha \leq 18 \\ (\pm c, \pm c, \pm c), & 19 \leq \alpha \leq 26 \end{cases} \quad (1)$$

ここで、 $c$  は定数で、格子間隔と時間刻みの比 ( $c = \Delta x / \Delta t$ ) と定義される。

格子ボルツマン法では、流体の粘性は仮想粒子どうしの衝突によって生じると仮定される。衝突項には、広く用いられている BGK (Bhatnagar-Gross-Krook) 衝突緩和モデル [17] を適用する。このとき、格子ボルツマン法の支配方程式は次のように表される。

$$f_{\alpha}(\mathbf{x} + \mathbf{c}_{\alpha} \Delta t, t + \Delta t) - f_{\alpha}(\mathbf{x}, t) = -\frac{1}{\tau} (f_{\alpha}(\mathbf{x}, t) - f_{\alpha}^{eq}(\mathbf{x}, t)) + F_{\alpha} \quad (2)$$

ここで、 $f_{\alpha}$  は速度  $\mathbf{c}_{\alpha}$  を持つ仮想粒子の速度分布関数である。  $\tau$  は BGK モデルにおける緩和時間係数で、以下の式により求められる。

$$\tau = \frac{1}{2} + \frac{3\nu}{c^2 \Delta t} \quad (3)$$

$\nu$  は流体の動粘度である.  $f_\alpha^{eq}$  は速度分布関数の平衡分布で, 次のように定義される.

$$f_\alpha^{eq} = w_\alpha \rho \left( 1 + \frac{3c_\alpha \cdot \mathbf{u}}{c^2} + \frac{9(c_\alpha \cdot \mathbf{u})^2}{2c^4} - \frac{3\mathbf{u} \cdot \mathbf{u}}{2c^2} \right) \quad (4)$$

ただし,  $w_\alpha$  は各速度方向の重みづけ係数で, 以下のよう な値を持つ.

$$w_\alpha = \begin{cases} 8/27, & \alpha = 0 \\ 2/27, & 1 \leq \alpha \leq 6 \\ 1/54, & 7 \leq \alpha \leq 18 \\ 1/216, & 19 \leq \alpha \leq 26 \end{cases} \quad (5)$$

また,  $\rho, \mathbf{u}$  は流体の密度, 流速であり, 以下のように速度分布関数の 0 次, 1 次モーメントをとることにより求められる.

$$\rho = \sum_{\alpha=0}^{26} f_\alpha \quad (6)$$

$$\mathbf{u} = \frac{1}{\rho} \sum_{\alpha=0}^{26} c_\alpha f_\alpha \quad (7)$$

$F_\alpha$  は外力項であり, 体積力によって生じる加速度  $\mathbf{a}$  を用いて次のように求められる.

$$F_\alpha = w_\alpha \rho \frac{3c_\alpha \cdot \mathbf{a}}{c^2} \Delta t \quad (8)$$

格子ボルツマン法の計算は, 並進ステップと衝突ステップに分かれる. 並進ステップは 27 個それぞれの速度分布関数を速度に応じた上流点から移流させる計算で, 静的に決められたパターンで隣接点参照を行う. 並進ステップでは, 速度分布関数の値は 1 ステップにつき 1 度だけ参照されるため, キャッシュメモリが有効に働かない. 一方で, 衝突ステップは, 同一格子点上の 27 個の速度分布関数の相互作用計算であり, 隣接格子点情報をいっさい参照しないため, 単純な実装でも十分に高い実行性能を出すことができる.

## 2.2 適合格子細分化法 (AMR)

AMR の構造には, 図 2 に示すような木データ構造で表される再帰的な領域分割によって生成される細分化格子を用いる. 3 次元格子の場合, 1 回の領域分割によって元の領域は 8 つに分割され, octree データ構造となる. Octree の各節点 (図 2(a) において白色の丸, および図 2(b) において白色の丸または色付きの四角の記号で表される要素) をノードと呼ぶ. 上下に実線でつながれたノードどうしは親子関係にあるといい, 上側のノードを親ノード, 下側のノードを子ノードと呼ぶ. 親ノードが共通であるような 2

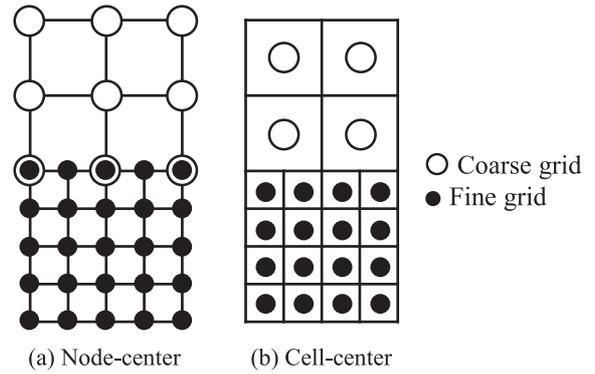


図 5 ノードセンタおよびセルセンタの格子配置  
Fig. 5 Lattice alignment of node-center and cell-center grid.

つ以上のノードは互いに兄弟ノードであるという. 子を持たないノードはリーフと呼ばれる. さらに本論文では, 1 個以上の子がリーフであるようなノードが特別な意味を持つため, これをマザーリーフと名付ける.

それぞれのリーフには一定数の格子点で構成される等間隔格子が割り当てられる. リーフの格子配置には, 図 5 に示すように, グリッド線の交点上に値を配置するノードセンタと, グリッド線が囲う領域の中心に値を配置するセルセンタの 2 つの方式がある. 本論文で用いているノードセンタの格子配置は, 解像度差 (レベル差) のあるリーフが隣接する位置において粗い格子の位置に必ず細かい格子が重複して存在しているため, 空間補間の精度が良い. ノードセンタを用いると, 格子点数は各空間軸の方向に 2 の累乗+1 個となる. また, 局所的に要求される解像度の適合性から, リーフの格子点数は大きすぎない (無駄に細かい格子を割り当てない) ことが求められる. このため本論文では, 各リーフにノードセンタで  $17^3$  の数の格子点を割り当てることにする.

Octree のリーフのノード深さを, 細分化レベル, または単にレベルと称し, リーフの格子解像度を表す指標とする. 最も粗い格子を持つリーフのレベルを 0 とし, レベルが 1 大きくなるごとにリーフの格子解像度を 2 倍にする. たとえば, 図 2(a) では, 最も粗いリーフは 2, 3, 4, 最も細かいリーフは 5, 6, 7, 8 であり, 全体で 2 倍の解像度 (格子間隔) の違いがある. 図 2(b) の例では, 最も細かい橙色リーフの細分化レベルは 2 で, 全体で 4 倍の格子解像度の違いがある. なお, AMR による計算の安定性を確保するため, 互いに隣接するリーフのレベル差は 0 または 1 に限定される.

## 3. AMR の GPU 実装における高速化手法

### 3.1 Naïve な実装

「Naïve な実装」とは, 「高速化などをまったく考慮せず, アルゴリズムのとおり単純にプログラミングした実装」を意味する. GPU カーネル関数を 1 種類だけ記述し, 1 格

子点の計算に1スレッドを割り当てて計算を行う。GPUカーネル関数のレジスタがあふれないよう、ブロック内のスレッド数のみ調整している。リーフには複数の格子点が割り当てられているため、リーフの内側（内部格子）と外殻部分（外殻格子）で隣接点参照のアクセスパターンが異なる。内部格子は、全計算空間に対して等間隔格子で計算する場合と同じ単純なインデックス計算により隣接点を参照することができる。一方、外殻格子では隣接点の一部が自身のリーフではなく隣接リーフに存在する。隣接リーフの格子点を参照するためにはインデックス計算に加えて隣接リーフの探索を行う必要があり、メモリアクセスの回数が増加する。また、隣接格子点のメモリアドレスが不連続になり、アクセスパターンも連続的でなくなる。隣接リーフにレベル差がある場合には、隣接リーフの格子が自身よりも粗いときと細かいときとでさらに処理が分岐する。隣接リーフのレベルが低い（格子が粗い）場合には、隣接点の位置に格子点が存在しないため、近傍の8点から値を補間する。隣接リーフのレベルが高い場合は隣接点の位置に必ず格子点が存在するため、その格子点の値を参照する。したがって、外殻格子では多数の条件分岐を含んだ計算となる。

NVIDIA社のCUDAフレームワークに基づいたGPU計算では、Warpと呼ばれる32個のスレッドがSIMT (Single Instruction Multiple Thread) で実行される。同一Warp内のスレッドが条件分岐によって互いに異なる処理を行う場合、Warp divergenceと呼ばれる冗長な命令実行が起こる。本AMRではリーフの格子点数が $17^3$ と小さいため、ほとんどのWarpに内部格子だけでなく数個の外殻格子が含まれる。このとき、内部格子の計算を担当するスレッドは自身のWarp内の外殻格子の計算が完了するまで待機することになる。それぞれの外殻格子が異なるアクセスパターンを持つ場合には、外殻格子をアクセスパターンごとに分けて計算する必要があり、スレッドの待機時間はさらに長くなる。

ほとんどのWarpは、28個の内部格子と4個の外殻格子の計算を行う。たとえば、外殻格子点が $x$ 軸方向の境界面にある場合、アクセスパターンは $x$ 軸の正方向の（右）面の格子と $x$ 軸の負方向の（左）面の格子で2つに分かれる。このため、1Warpの計算は、内部格子、左面の外殻格子、右面の外殻格子の3種類の処理に分岐する。条件分岐が多くなると、待機状態のスレッドの数が増加し、命令を実行しているスレッド（アクティブスレッド）の数が減少する。外殻格子の計算を行っている間のアクティブスレッド数は1Warp（32スレッド）あたりわずか2スレッドであり、残りの30スレッドが待機状態となるため、並列化効率が非常に低くなる。

### 3.2 内部格子と外殻格子のカーネル分割

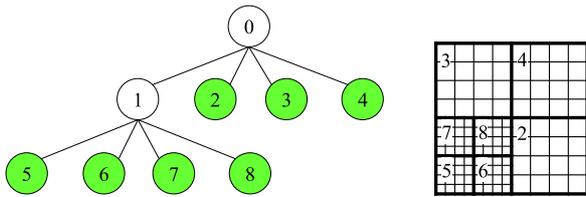
外殻格子の計算は内部格子に比べて低速で、内部格子と外殻格子の計算スレッドの間で発生するWarp divergenceは深刻な実行性能低下を招く。Warp divergenceの影響を低減させるため、内部格子と外殻格子を異なるカーネル関数に分割して計算する。これにより、内部格子に対する計算カーネルでは実行性能が著しく向上する。さらに内部格子の計算を高速化するため、リーフ内格子のメモリアドレスの並べ替えによるコアレスニングの改善、スレッドの冗長化やWarp Shuffleを利用した完全なコアレスアクセスも実装している[18]。一方、外殻格子の計算カーネルでは外殻格子点の計算スレッドどうしの間でWarp divergenceが発生するため、外殻格子の計算は低速のままである。

### 3.3 マザーリーフ法による外殻格子点数の削減

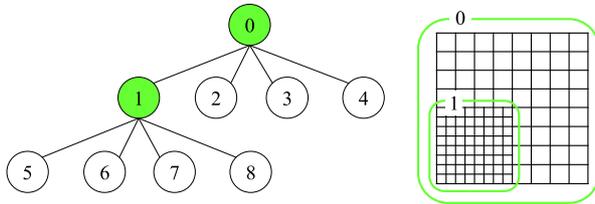
リーフの外殻格子は隣接点参照のメモリアクセスパターンが悪く、高い実行性能が得られない。外殻格子の数が内部格子に比べて相対的に少ないほど、計算の実行性能は高くなる。リーフあたりの外殻格子の数は、3次元の場合リーフの格子点数の3分の2乗に比例するため、リーフの格子点数を $17^3$ から $33^3$ 、 $65^3$ などと増加させることで外殻格子の数を相対的に減少させることができる。しかし、リーフの格子点数が増加すると細分化の徹密さが低下する（無駄に細かい格子を多く使う）ことにつながるため、AMR法の実装として適していない。

AMRの格子適合性を維持しつつ外殻格子の数を削減する手法として、マザーリーフ法を提案する。マザーリーフ法では、本来末端のリーフが持つべき格子を、兄弟リーフでまとめて共通の親ノード（マザーリーフ）に持たせる。兄弟ノードの中にリーフと単なるノードの両方が存在する場合には、単なるノードにも仮の格子を割り当て、マザーリーフでまとめて管理する。簡単のため、図6に示す2次元の例を用いて説明する。(a)はリーフに格子を割り当てようとする元のデータ構造、(b)はマザーリーフ法を用いたときのデータ構造である。木構造において緑色で示したノードに対して格子を割り当てている。図6のようにノード2, 3, 4, 5, 6, 7, 8がリーフであるような場合、図6(b)のようにリーフ5, 6, 7, の格子をマザーリーフ1で管理し、リーフ2, 3, 4の格子およびノード1の仮の格子をマザーリーフ0で管理する。このとき、マザーリーフ0, 1の重複する領域では複数の解像度の格子が重なって存在しているが、細かい格子（マザーリーフ1）から粗い格子（マザーリーフ0）へのとき、空間補間を行い、粗い格子の値を補間値で上書きする。

マザーリーフ法は細分化格子の構造を変化させることなく外殻格子数を削減することができ、連続アドレスで計算できる格子点数を増加させることができる。計算領域の総格子点数は粗い格子と細かい格子が重なり合う分だけ増大す



(a) Traditional data structure.



(b) Data structure with Mother-Leaf Method.

図 6 マザーリーフ法 (2次元の例)

Fig. 6 Mother-Leaf Method (2D example).

るが、総格子点数の増大率は、兄弟ノードのうち7つのノードに子ノードを生成するような領域分割を繰り返した場合でも最大で 4.50% ( $\lim_{d \rightarrow \infty} (\sum_{l=0}^{d-1} 7^l + 7^d \times 8) / \sum_{l=0}^d 7^l$ ) であり、冗長な計算に対するメモリ使用量や計算量は小さい。マザーリーフの格子点数は、リーフの  $17^3$  に対して  $33^3$  となり、そのうち外殻格子点数はマザーリーフで 6,146 個 ( $33^3 - 31^3$ )、リーフで 1,538 個 ( $17^3 - 15^3$ ) である。マザーリーフ 1 つあたりの外殻格子点数はリーフに比べて 4 倍程度に増加するが、マザーリーフの総数はリーフの 8 分の 1 程度に減少するため、計算領域全体の外殻格子点数は 2 分の 1 程度に減少する。

Octree のリーフの 1 段上のノードを利用する手法がメモリアクセスの効率向上に有効であることは、Crassin らの CG レンダリング手法 [19] によってすでに示されている。しかし、本手法をステンシル計算に適用した例はなく、外殻格子点数の削減による高速化の効果も未知であるため、本論文においてその効果を確認する。

### 3.4 C++テンプレートによる外殻格子のカーネル分割

外殻格子には、格子点の位置によって複数のアクセスパターンが存在する。Naïve な実装ではそのアクセスパターンをカーネル内の条件分岐文で判定していた。多数の条件分岐があるため、条件分岐の評価に多大な計算時間が費やされて、アクセスパターンの異なる外殻格子が同一 Warp 内で計算されることによる Warp divergence も発生し、実行性能は著しく低下していた。

外殻格子計算の高速化のためには、存在しうるアクセスパターンをすべて列挙し、複数のカーネル関数に分割して記述する方法が考えられる。しかし、D3Q27 格子ボルツマン法においては、外殻格子点の位置によってパターンが 26 通りあることに加え、そのそれぞれに 27 通りの隣接点

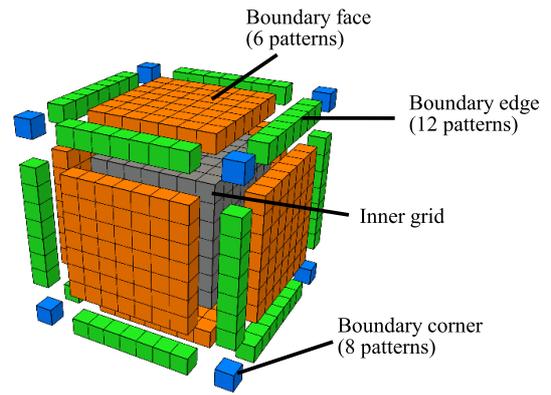


図 7 隣接点参照パターンに基づく格子の分類

Fig. 7 Classification of lattices based on stencil access patterns.

参照があるため、それらの組合せによるアクセスパターンの総数は 702 に達する。これほど多数のパターンを手動で記述することは、プログラムの生産性を考えると現実的でない。

本研究では、Harris の C++テンプレートを用いた GPU カーネル生成による最適化手法 [20] を応用し、プログラムの生産性を損なうことなく、外殻格子における 702 通りすべてのアクセスパターンを列挙する高速化手法を提案する。まず、格子点の位置を同一のアクセスパターンごとに分類する。リーフの格子点について、隣接点へのアクセスパターンごとの分類を図 7 に示す。リーフの格子点は次の 27 パターンに分類される。

$$(i, j, k) \in \{1, 2, \dots, m-2\}^3 \quad (9)$$

$$(i, j, k) \in \{0, m-1\} \times \{1, 2, \dots, m-2\}^2 \quad (10)$$

$$(i, j, k) \in \{0, m-1\}^2 \times \{1, 2, \dots, m-2\} \quad (11)$$

$$(i, j, k) \in \{0, m-1\}^3 \quad (12)$$

ここで、 $m^3$  はリーフあたりの格子点数であり、 $(i, j, k) \in \{0, 1, 2, \dots, m-1\}^3$  はリーフの格子点のインデックスである。式 (9) は内部格子、式 (10)–(12) は外殻格子である。外殻格子はさらに、式 (10) の 6 種類の面、式 (11) の 12 種類の辺、式 (12) の 8 種類の頂点に分けられる。面の格子は 1 つの隣接リーフを参照し、辺の格子は 3 つの隣接リーフへの参照を必要とする。頂点格子には 7 つの隣接リーフにアクセスするパターンが存在する。隣接点参照のパターンごとに分類した格子のパターンを示す変数として、 $gi, gj, gk$  を次のように定義する。

$$gi = \begin{cases} -1 & (i = 0) \\ 0 & (i = 1, 2, \dots, m-2) \\ 1 & (i = m-1) \end{cases} \quad (13)$$

Program 1 C++ template kernel for boundary grids.

```

template<int gi, int gj, int gk>
__global__ void leaf_bound(...) {
    /* 格子ボルツマン法の速度分布関数 */
    float f[27];
    /* リーフの番号および格子点のインデックスli,j,を計算k */
    const int l = ...;
    const int i = ..., j = ..., k = ...;
    /* 格子ボルツマン法の並進過程に基づく隣接点参照 */
    f[0] = read_stencil<gi, gj, gk, 0, 0, 0>(l, i, j, k, 0,...);
    f[1] = read_stencil<gi, gj, gk, 1, 0, 0>(l, i, j, k, 1,...);
    f[2] = read_stencil<gi, gj, gk, 0, 1, 0>(l, i, j, k, 2,...);
    ... /* 残り個の24関数も同様に呼び出すread_stencil */
    /* 格子ボルツマン法の衝突項を計算 */
    f[0] = ...;
    f[1] = ...;
    ...
    /* f[] の計算結果をグローバルメモリに書き込む */
    .....
}

```

Program 2 Detail of the subroutine “read\_stencil.”

```

template<int gi, int gj, int gk, int si, int sj, int sk>
__device__ float read_stencil(int l, int i, int j, int k, int alpha, ...) {
    float retval;
    const int ni = (gi == si) ? si : 0;
    const int nj = (gj == sj) ? sj : 0;
    const int nk = (gk == sk) ? sk : 0;
    if(ni != 0 || nj != 0 || nk != 0) {
        /* 隣接リーフの格子点を参照 */
        retval = read_stencil_neighbor_leaf<ni, nj, nk>(l, i + si, j + sj, k + sk, alpha);
    } else {
        /* 自分自身のリーフの格子点を参照 */
        retval = fg[ idx(l, i + si, j + sj, k + sk, alpha) ];
    }
    return retval;
}

```

$$gj = \begin{cases} -1 & (j = 0) \\ 0 & (j = 1, 2, \dots, m-2) \\ 1 & (j = m-1) \end{cases} \quad (14)$$

$$gk = \begin{cases} -1 & (k = 0) \\ 0 & (k = 1, 2, \dots, m-2) \\ 1 & (k = m-1) \end{cases} \quad (15)$$

$gi, gj, gk$  をテンプレート引数とし、Program 1 のような GPU カーネル関数を記述する。各スレッドの担当するリーフの番号や格子点のインデックスは、 $gi, gj, gk$  およびスレッド番号 (threadIdx, blockIdx) の組合せによって定める。Program 1 のカーネル関数を、

```

leaf_bound<-1,0,0><<<gridDim,blockDim>>>(…),
leaf_bound<1,0,0><<<gridDim,blockDim>>>(…)

```

などと、 $gi, gj, gk$  を変化させて呼び出すことで、C++ テンプレートの機能によってそれぞれのアクセスパターンに対応する 26 通りのカーネル関数が生成される。なお、 $\text{leaf\_bound}<0,0,0>$  は内部格子に対応しており、実際には使用しない。

$\text{read\_stencil}$  は隣接点の値を参照するデバイス関数である。 $gi, gj, gk$  に加えて隣接点の方向を表す変数  $(si, sj, sk) \in \{-1, 0, 1\}^3$  をテンプレート引数にとり、Program 2 のように記述する。 $(gi, gj, gk)$  の組合せは 26 通り、 $(si, sj, sk)$  の組合せは 27 通りあり、全部で 702 通りのアクセスパターンに対応する  $\text{read\_stencil}$  関数がテンプレート展開によって生成される。また、 $\text{read\_stencil\_neighbor\_leaf}$  は、隣接リーフの方向  $(ni, nj, nk)$  に従って隣接リーフの探索および格子の値の参照を行うサブルーチンである。idx はリーフ内の格子を参照

するためのインデックス計算を行う関数である。fg はグローバルメモリ上に記憶されている速度分布関数の値へのポインタである。Program 2 における if 文は隣接点がリーフ内にあるかどうかの判定であり、隣接点がリーフ外（隣接リーフ）にあるとき真、隣接点がリーフ内にあるとき偽となる。格子ボルツマン法の場合は条件分岐がすべて静的であるため、以上のようなアクセスパターンを適切に分類したテンプレート関数がコンパイル時に展開される。これにより、条件分岐の評価に要する時間や Warp divergence による実行性能の低下がなくなり、実行性能の向上を図ることができる。

カーネル関数を 26 通りに分割したことでそれぞれのカーネルが担当する格子点数が少なくなっており、単一のカーネルでは GPU 計算に十分な並列数をとることができない。そこで、CUDA のストリームを用いて複数のカーネルを同時実行することで並列数を確保する。ストリームの実行はランタイム時に動的にスケジューリングされるが、表 1 に示す実行環境において最大で 11 個のカーネルが同時実行されることを確認している。

#### 4. 実行性能測定

AMR を導入した格子ボルツマン法に 3 章で説明した高

表 1 実行環境

Table 1 Execution environment.

CPU	Intel Xeon X5670 (2.93 GHz)
Main memory	54 GB
Operating system	SUSE Linux Enterprise Server 11 SP3
GPU	NVIDIA K20Xm
C++ Compiler	GNU Compiler Collection 4.3.4
CUDA driver	version 6.5

表 2 計算条件（マザーリーフ法なし）

Table 2 Computational condition without mother-leaf method.

mesh size per leaf	4,913 (17 <sup>3</sup> )
number of leaves	512
number of total grids	2,515,456
(number of inner lattices)	(1,728,000)
(number of outer shell lattices)	(787,456)
type of floating point number	single precision

速化手法を適用し、実行性能測定を行った。実行環境を表 1 に示し、計算条件を表 2、表 3 に示す。3.3 節で説明したマザーリーフ法の適用により、同一の問題であっても総格子点数、内部格子点数および外殻格子点数が変化するため、マザーリーフ法あり・なし両方の計算条件を示している。空間補間や境界条件などの特殊な処理を含まないテンシル計算の部分のみで性能を測定するため、全計算領域でリーフの細分化レベルは一定、境界条件は周期境界条件とした。また、浮動小数点演算はすべて単精度とした。

チューニングを行っていない Naïve な実装に対して、3 章で説明した高速化手法を 1 つずつ加える形で適用していき、1 タイムステップの計算時間を測定することで、それぞれの手法の効果を確かめた。計算時間は、内部格子、外殻格子、全格子合計（内部+外殻）のそれぞれについて測定した。計算時間の測定には CUDA プロファイラ（nvprof および NVIDIA Visual Profiler）を用い、内部格子カーネルの実行時間、および 26 個の外殻格子カーネルのうち最初のカーネルの起動から最後のカーネルの実行完了までの時間を測定した。なお、GPU カーネルの起動にかかる時間は 1 カーネルあたり数百ナノ秒と非常に短いため、測定から除外した。

#### 4.1 実行時間の測定

計算時間の測定結果を表 4 に示す。Inner は内部格子の計算時間、Outer Shell は外殻格子の計算時間であり、Total は合計の計算時間を表している。表 4 の [3.1] は高速化手法を適用していない Naïve な実装を指している。Naïve な実装では内部格子と外殻格子を同一カーネルで計算していたため、合計の計算時間のみを示している。[3.2], [3.3], [3.4] は適用した高速化手法を表しており、その数字は 3 章の節番号に対応している。

表 3 計算条件（マザーリーフ法あり）

Table 3 Computational condition with mother-leaf method.

mesh size per mother-leaf	35,937 (33 <sup>3</sup> )
number of mother-leaves	64
number of total grids	2,299,968
(number of inner lattices)	(1,906,624)
(number of outer shell lattices)	(393,344)
type of floating point number	single precision

表 4 それぞれの高速化手法の適用前後の計算時間

Table 4 Computation time on each tuning applied.

	Inner		Outer Shell		Total	
	time [msec]	speedup	time [msec]	speedup	time [msec]	speedup
[3.1]	—	—	—	—	142.187	—
[3.2]	2.642	—	23.731	—	26.373	5.39
[3.3]	2.705	0.98	12.111	1.96	14.816	1.78
[3.4]	2.705	1.00	5.270	2.29	7.975	1.86

内部格子と外殻格子のカーネル分割 [3.2] によって 5.39 倍の高速化を達成した。外殻格子の計算に内部格子の 8.98 倍程度の時間を要しており、外殻格子は計算効率が非常に低いことが分かる。

マザーリーフ法 [3.3] の適用により、外殻格子の格子点数が削減され、外殻格子の計算を 1.96 倍に高速化することができた。内部格子は、格子点数が増加したために計算時間もわずかに増大しているが、外殻格子の高速化の方が効果は大きく、全体でも 1.78 倍の高速化を達成した。

C++テンプレートによるカーネル分割 [3.4] により、外殻格子の計算速度を 2.29 倍に向上させることができた。内部格子の計算には変化がなく、全体としては 1.86 倍の高速化となった。

以上の高速化手法の適用により、Naïve な実装に対して合計で 17.8 倍の高速化を達成した。特に 2 度の計算カーネルの分割 ([3.2] および [3.4]) で、計算速度をそれぞれ 5.39 倍および 1.86 倍に高速化した点は、AMR 法に特有の新しい高速化手法といえる。

## 4.2 Roofline モデルによる評価

4.1 節におけるすべての高速化手法を適用し、実行性能の上限値を予測する Roofline モデル [21] を用いて評価する。計算の所要時間に対しメモリアクセスの時間と浮動小数点演算時間の長いほうを選ぶ単純なモデルである。

$$P_{\text{reachable}} = \min(P_{\text{peak}}, B_{\text{peak}} \times I), \quad (16)$$

$$I = \frac{F}{B}$$

ここで、 $P_{\text{peak}}$  は浮動小数点演算器のピーク性能、 $B_{\text{peak}}$  はメモリバンド幅のカタログ性能である。 $F$  は 1 格子点あたりの浮動小数点演算の回数、 $B$  は 1 格子点あたりのメモリアクセス量である。また、 $I$  は演算強度と呼ばれる値で、浮動小数点演算の回数とメモリアクセス量の比として与えられる。本計算条件においては、

$$P_{\text{peak}} = 3953.23 \text{ GFLOPS}, B_{\text{peak}} = 249.6 \text{ GB/sec},$$

$$F = 377 \text{ flop}, B = 220 \text{ byte}, I = 1.713$$

であり、したがって実行性能の上限値は  $P_{\text{reachable}} = 427.6$  GFLOPS と予測される。

$P_{\text{peak}}/B_{\text{peak}} = 15.8$  に対して演算強度が  $I = 1.713$  ということから、格子ボルツマン法はメモリアクセスが律速の計算であることが分かる。以下の改良型 Roofline モデル [22] を用いると上限値の予測精度を少し改善することができる。

$$P_{\text{reachable}} = \frac{F}{F/P_{\text{peak}} + B/B_{\text{peak}}}$$

$$= \frac{I}{I + P_{\text{peak}}/B_{\text{peak}}} P_{\text{peak}} \quad (17)$$

実行性能の上限値は  $P_{\text{reachable}} = 385.8$  GFLOPS と予測さ

表 5 実行性能

Table 5 Results of performance.

	Performance [GFLOPS]	Ratio [%]
Reachable	385.8	—
Inner	265.7	68.9
Outer Shell	28.14	7.29
Total	108.7	28.2

れる。改良型 Roofline モデルによる実行性能の上限値、ならびに、内部格子、外殻格子および全格子合計の実行性能を表 5 に示す。内部格子の実行性能は上限値の 68.9% であり、 $33^3$  という小さいマザーリーフの格子を GPU で計算するという厳しい条件の下では、十分に最適化された性能であるといえる。また、合計の実行性能が上限値の 28.2% であることから、AMR の格子点数が等間隔格子の格子点数に対して 28.2% よりも少なければ、AMR の方が等間隔格子よりも高速に計算を行うことができることが分かる。すなわち、AMR によって格子点数を 4 分の 1 以下に削減することで、メモリ使用量だけでなく計算速度においても AMR が優位となる。

## 5. 応用計算

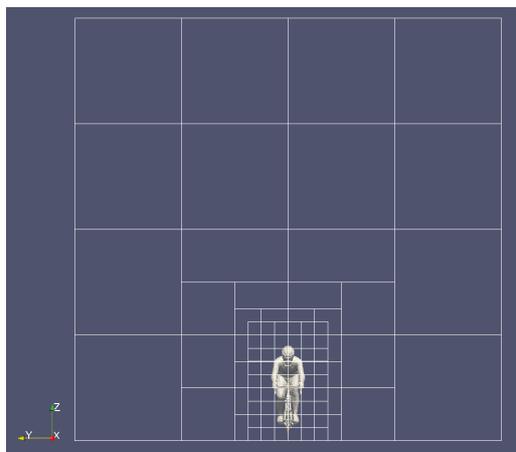
本論文で提案する高速化手法を適用した例として、自転車競技の選手の周りの流れの計算を行った。計算領域の中で自転車競技の選手と自転車は静止した物体として扱い、自転車の進行方向から一定速度で風が流入すると仮定している。このような空力計算では、計算境界において圧力波が反射し、計算精度を悪化させることが知られており、精度良く計算するためには計算領域を物体の遠方までとる必要がある。等間隔格子では広い計算領域をとるために物体近傍で格子解像度が不足するおそれがあるが、AMR を用いることで物体近傍を局部的に高解像度にするができる。

計算領域に割り当てた AMR の格子配置を図 8 に示す。白枠線はマザーリーフの境界線を表しており、各マザーリーフには  $33^3$  の格子が割り当てられている。AMR の最深の細分化レベルを 3 とし、物体近傍に高解像度格子を集中させた。最高の格子解像度は、計算領域全体に  $1025^3$  の等間隔格子を割り当てた解像度に相当し、格子点数は、 $1025^3$  の等間隔格子に比べて 5% にまで削減されている。なお、細分化レベルの異なる格子間の補間処理に要する時間は全体の計算時間に対してわずかであるため、補間処理に対するチューニングは特に行っていない。

本計算では、流体の計算と同時に速度場に沿って移流するトレーサ粒子（パッシブ・スカラー粒子とも呼ばれる）を計算し、計算結果を可視化している。乱流が十分に発達した時刻のスナップショットを図 9 に示す。レイノルズ数 40 万の流れになっているため、衝突ステップに



(a) Side view of the computational area.



(b) Front view of the computational area.

図 8 複雑形状物体周りの流れの計算領域

Fig. 8 Computational area for a flow around a complex shape body.



図 9 複雑形状物体周りの流れの計算結果

Fig. 9 Result of a calculation for a flow around a complex shape body.

Multiple Relaxation Time (MRT) モデル [23] を適用し、さらに LES モデルとしてコヒーレント構造スマゴリンスキーモデル (CSM) を導入している [2], [24]. 同じ位置から発生させているトレーサ粒子が不規則に乱れていて、妥当な乱流速度場が計算されていることが分かる。

本計算は 1 台の GPU (NVIDIA K80: メモリ 12 GB) で

行っており、時間ステップ数は約 15 万、トレーサ粒子の計算やファイルの出力などを含めない場合の全実行時間は約 14 時間であった。AMR を適用しない場合、1 台の GPU で  $1025^3$  の等間隔格子を計算するには 248.7 GB のメモリが必要になる。1 台の GPU では計算できないために実行時間を直接計測することができないが、 $1025^3$  の等間隔格子で計算した場合の実行時間を改良型 Roofline モデルから予測することができる。ただし、MRT および CSM を導入したため、演算量  $F$  とメモリアクセス量  $B$  の値は 4 章とは異なる。本計算はすべて単精度浮動小数点数で計算しており、

$$P_{\text{peak}} = 2.8 \text{ TFLOPS}, B_{\text{peak}} = 240 \text{ GB/sec},$$

$$F = 3056 \text{ flop}, B = 468 \text{ byte}$$

である。式 (17) による実行性能の上限値は  $P_{\text{reachable}} = 1.0$  TFLOPS である。 $P_{\text{reachable}}$  を用いると、 $1025^3$  の等間隔格子で 15 万ステップを計算した場合の実行時間は以下のように予測できる。

$$\begin{aligned} T_{\text{estimated}} &= \frac{3056 \times 1025^3 \times 150,000 \text{ [flop]}}{1.0 \times 10^{12} \text{ [flop/sec]}} \\ &= 4.9 \times 10^5 \text{ sec} \\ &\approx 136 \text{ hour} \end{aligned}$$

したがって、AMR 法の適用とその高速化により、全計算領域を等間隔格子で計算する場合とほぼ同じ計算結果を得ることに對して、Time-to-Solution で 9.7 倍の高速化を達成したといえる。AMR の細分化の深度を上げれば物体近傍を物体形状に適合して細分化することができ、等間隔格子との計算時間の差はますます広がる。

## 6. おわりに

本研究では、ステンシル計算の一種である格子ボルツマン法に適合細分化格子法 (AMR) を導入した GPU 計算の高速化手法を提案した。AMR のリーフに割り当てられる格子は等間隔直交格子に比べて小さいため、GPU で高効率な並列計算を行うことが難しい。特に、リーフの境界部分にある外殻格子は隣接参照のために他のリーフの格子を参照する必要があり、アクセスパターンが悪く高い実行性能が出ない。外殻格子点を削減するため、Octree のリーフの 1 段上のノードに格子を割り当てる「マザーリーフ法」を提案し、外殻格子計算で 1.96 倍 (全体で 1.78 倍) の高速化を達成した。さらに、外殻格子計算では隣接格子へのアクセスパターンを適切に分類して実装する必要があるが、D3Q27 格子ボルツマン法による 3 次元計算の場合、リーフの外殻格子のアクセスパターンは 702 通りになり、すべてのパターンを手動で列挙して実装することはプログラムの生産性や保守性の観点から現実的でない。これまで格子ボルツマン法や AMR には適用されていない「C++テンプレートによる GPU カーネル関数の生成」の方法により、

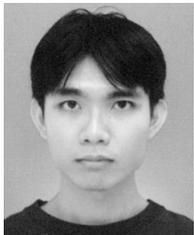
実行性能の低い外殻格子計算を 2.29 倍に高速化した (全体でも 1.86 倍の高速化)。1 種類のためのテンプレート関数を記述し、テンプレート引数を変化させコンパイル時にアクセスパターンを判定をすることで、プログラムの生産性を維持しながら計算を高速化することができた。本手法は D3Q27 以外の D3Q15 や D3Q19 格子ボルツマン法にも適用できるが、D3Q27 より条件分岐の数が少ないため、高速化率は低下することが予想される。また、計算中のデータに応じた条件分岐でなければ、他のステンシル計算の一部に適用し高速化できる可能性がある。

謝辞 本研究の一部は科学研究費補助金・基盤研究 (S) 課題番号 26220002 「ものづくり HPC アプリケーションのエクサスケールへの進化」、科学技術振興機構 CREST 「ポストペタスケール高性能計算に資するシステムソフトウェア技術の創出」から支援をいただいた。記して謝意を表す。

#### 参考文献

- [1] Shimokawabe, T., Aoki, T., Takaki, T., Endo, T., Yamanaka, A., Maruyama, N., Nukada, A. and Matsuoka, S.: Peta-scale Phase-field Simulation for Dendritic Solidification on the TSUBAME 2.0 Supercomputer, *Proc. 2011 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11*, New York, NY, USA, ACM, (online), DOI: 10.1145/2063384.2063388 (2011).
- [2] 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).
- [3] 下川辺隆史, 青木尊之, 小野寺直幸: 複数 GPU による格子に基づいたシミュレーションのためのマルチ GPU コンピューティング・フレームワーク, 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.2014, pp.78-86 (2013).
- [4] Wang, X. and Aoki, T.: Multi-GPU Performance of Incompressible Flow Computation by Lattice Boltzmann Method on GPU Cluster, *Parallel Computing*, Vol.37, No.9, pp.521-535 (online), DOI: 10.1016/j.parco.2011.02.007 (2011).
- [5] Rahimian, A., Lashuk, I., Veerapaneni, S., Chandramowlishwaran, A., Malhotra, D., Moon, L., Sampath, R., Shringarpure, A., Vetter, J., Vuduc, R., Zorin, D. and Biros, G.: Petascale Direct Numerical Simulation of Blood Flow on 200K Cores and Heterogeneous Architectures, *Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10*, Washington, DC, USA, IEEE Computer Society, (online), DOI: 10.1109/SC.2010.42 (2010).
- [6] Grinberg, L., Morozov, V., Fedosov, D., Insley, J., Papka, M., Kumaran, K. and Karniadakis, G.: A new computational paradigm in multiscale simulations: Application to brain blood flow, *Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pp.1-12 (2011).
- [7] Performance modeling and analysis of heterogeneous lattice Boltzmann simulations on CPU — GPU clusters, *Parallel Computing*, Vol.46, pp.1-13 (online), DOI: <http://dx.doi.org/10.1016/j.parco.2014.12.003> (2015).
- [8] Rossinelli, D., Hejazialhosseini, B., Hadjidoukas, P., Bekas, C., Curioni, A., Bertsch, A., Futral, S., Schmidt, S.J., Adams, N.A. and Koumoutsakos, P.: 11 PFLOP/s Simulations of Cloud Cavitation Collapse, *Proc. 2013 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '13*, New York, NY, USA, ACM, (online), DOI: 10.1145/2503210.2504565 (2013).
- [9] Shimokawabe, T., Aoki, T. and Onodera, N.: High-productivity Framework on GPU-rich Supercomputers for Operational Weather Prediction Code ASUCA, *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis, SC'14*, Piscataway, NJ, USA, pp.251-261, IEEE Press (online), DOI: 10.1109/SC.2014.26 (2014).
- [10] Berger, M. and Oliger, J.: Adaptive mesh refinement for hyperbolic partial differential equations, *Journal of Computational Physics*, Vol.53, No.3, pp.484-512 (online), DOI: [http://dx.doi.org/10.1016/0021-9991\(84\)90073-1](http://dx.doi.org/10.1016/0021-9991(84)90073-1) (1984).
- [11] Olga, F. and Dieter, H.: Grid Refinement for Lattice-BGK Models, *Journal of Computational Physics*, Vol.147, No.1, pp.219-228 (online), DOI: <http://dx.doi.org/10.1006/jcph.1998.6089> (1998).
- [12] Rosenberg, D., Fournier, A., Fischer, P. and Pouquet, A.: Geophysical-astrophysical spectral-element adaptive refinement (GASpAR): Object-oriented h-adaptive fluid dynamics simulation, *Journal of Computational Physics*, Vol.215, No.1, pp.59-80 (online), DOI: <http://dx.doi.org/10.1016/j.jcp.2005.10.031> (2006).
- [13] Tu, T., O'Hallaron, D. and Ghattas, O.: Scalable Parallel Octree Meshing for TeraScale Applications, *Supercomputing, 2005, Proc. ACM/IEEE SC 2005 Conference*, p.4 (online), DOI: 10.1109/SC.2005.61 (2005).
- [14] Burstedde, C., Ghattas, O., Gurnis, M., Isaac, T., Stadler, G., Warburton, T. and Wilcox, L.: Extreme-Scale AMR, *Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC'10*, Washington, DC, USA, IEEE Computer Society, (online), DOI: 10.1109/SC.2010.25 (2010).
- [15] Wu, J., Lan, Z., Xiong, X., Gnedin, N. and Kravtsov, A.: Hierarchical Task Mapping of Cell-based AMR Cosmology Simulations, *Proc. International Conference on High Performance Computing, Networking, Storage and Analysis, SC '12*, Los Alamitos, CA, USA, pp.75:1-75:10 IEEE Computer Society Press (online), available from (<http://dl.acm.org/citation.cfm?id=2388996.2389098>) (2012).
- [16] Malhotra, D., Gholami, A. and Biros, G.: A Volume Integral Equation Stokes Solver for Problems with Variable Coefficients, *Proc. 2015 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC'14*, Washington, DC, USA, IEEE Computer Society, pp.92-102 (online), DOI: 10.1109/SC.2014.13 (2014).
- [17] Bhatnagar, P.L., Gross, E.P. and Krook, M.: A Model for Collision Processes in Gases. I. Small Amplitude Processes in Charged and Neutral One-Component Systems, *Physical Reviews*, Vol.94, pp.511-525 (online), DOI: 10.1103/PhysRev.94.511 (1954).
- [18] 長谷川雄太, 青木尊之: ステンシル計算の高速化のための C++テンプレートによる GPU カーネル生成, 情報処

- 理学会研究報告, Vol.2015-HPC-149, No.13 (2015).
- [19] Crassin, C., Neyret, F., Lefebvre, S. and Eisemann, E.: GigaVoxels: Ray-guided Streaming for Efficient and Detailed Voxel Rendering, *Proc. 2009 Symposium on Interactive 3D Graphics and Games, I3D '09*, New York, NY, USA, ACM, pp.15-22 (online), DOI: 10.1145/1507149.1507152 (2009).
- [20] Harris, M.: High Performance Computing with CUDA; Optimizing CUDA, *Tutorial of 2007 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC), SC'07* (2007).
- [21] Williams, S., Waterman, A. and Patterson, D.: Roofline: An Insightful Visual Performance Model for Multicore Architectures, *Comm. ACM*, Vol.52, No.4, pp.65-76 (online), DOI: 10.1145/1498765.1498785 (2009).
- [22] Shimokawabe, T., Aoki, T., Muroi, C., Ishida, J., Kawano, K., Endo, T., Nukada, A., Maruyama, N. and Matsuoka, S.: An 80-Fold Speedup, 15.0 TFlops Full GPU Acceleration of Non-Hydrostatic Weather Model ASUCA Production Code, *Proc. 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC), SC'10*, pp.1-11 (online), DOI: 10.1109/SC.2010.9 (2010).
- [23] Suga, K., Kuwata, Y., Takashima, K. and Chikasue, R.: A D3Q27 multiple-relaxation-time lattice Boltzmann method for turbulent flows, *Computers & Mathematics with Applications*, Vol.69, No.6, pp.518-529 (2015).
- [24] Kobayashi, H.: The subgrid-scale models based on coherent structures for rotating homogeneous turbulence and turbulent channel flow, *Physics of Fluids*, Vol.17, No.4, p.045104 (2005).



長谷川 雄太

1991年生。2014年香川高等専門学校専攻科創造工学専攻修了。2016年東京工業大学大学院総合理工学研究科創造エネルギー専攻修士課程修了。2016年同大学工学院機械系機械コース博士課程進学。



青木 尊之 (正会員)

1960年生。1983年東京工業大学理学部応用物理学科卒業。1985年同大学大学院総合理工学研究科エネルギー科学専攻修了。1985年富士通研究所厚木研究所入社。1986年東京工業大学大学院総合理工学研究科助手。1997年同大学原子炉工学研究所助教授。2001年同大学学術国際情報センター教授。数値流体力学, 計算力学, GPU コンピューティングの研究に従事。文部科学大臣表彰, 日本応用数学会業績賞, ACM ゴードンベル賞ほか。日本機械学会フェロー, 日本応用数学会等会員。