

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

長谷川 雄太^{1,a)} 青木 尊之¹

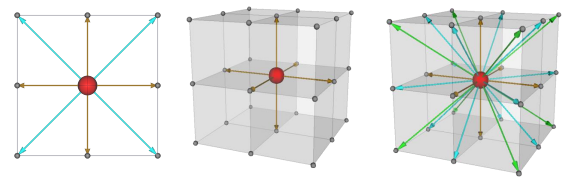
概要：ステンシル計算である格子ボルツマン法 (LBM) に適合格子細分化法 (AMR) を導入する際には、細分化の末端のリーフのサイズが小さいために GPU に実装した場合の計算効率が低い。特にリーフの境界部分はステンシルアクセスする格子点が方向によってリーフ内にある場合とリーフ外 (隣接リーフ) にある場合とがあり、アクセスのパターンを決定するために多数の条件分岐が必要となり、実行性能が大幅に低下する。そこで、すべてのアクセスパターンを列挙し、それぞれを別のカーネル関数に分離して記述する方法を提案する。プログラムの生産性を考え、C++テンプレートを用いることで、コンパイル時に 26 種類の GPU 計算のカーネル関数を生成し、実行性能とプログラムの生産性を両立させる。リーフ上の格子点の位置と隣接格子点の方向をテンプレート引数に取ることで if 文がコンパイル時に評価され、実行時に条件分岐を含まないカーネル関数を生成する。3 次元の 27 点ステンシル計算である D3Q27LBM において、本手法の適用によりリーフ境界の格子の計算を 2.29 倍高速化することができた。

1. はじめに

規則的に並んだ格子点データの値を隣接格子の数 n を用いて同じパターンで更新することを n 点ステンシル計算と呼ぶ。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 点ステンシル計算などがある。これらのステンシル計算は、模式的には図 1 のように表すことができる。ステンシル計算は、拡散方程式や移流方程式などの数値計算に応用されており、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伴い膨大な数の格子点を扱う大規模計算が可能となったことで、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) などのアクセラレータ (演算加速器) は、ステンシル計算に適しているため、さまざまな物理シミュレーションの大規模計算に用いられ、成功を収めている [1] [2]。

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直交格子が用いられてきた (SC13



(a) 9 points (2D), (b) 7 points (3D), (c) 27 points (3D)

図 1 n 点ステンシル計算の例

Fig. 1 Examples of n -points stencil computation

ゴードンベル賞 [3] を参照)。等間隔直交格子はステンシル計算におけるメモリアクセスの高い実行性能を得ることができる。しかしながら、物理シミュレーションにおいては全ての計算空間で高い解像度が要求されるわけではなく、等間隔直交格子は計算効率が低い。たとえば、自動車の空力解析では、物体近傍とその他の周辺の空間で必要な解像度に 100 倍以上の差がある。したがって、場所によって解像度を変更する手法を導入することで、計算機リソースの制約を克服し、現状よりも高精度かつ大規模な計算を行う必要がある。有限要素法や有限体積法では、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となり不規則になるため、ペタスケールを超える大規模計算には不向きと言われている。

等間隔直交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には、適合格子細分化法 (Adaptive Mesh Refinement; AMR 法)

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

^{a)} hasegawa@sim.gsfc.titech.ac.jp

が有効である。AMR 法は、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異なる解像度の格子を割り当てる手法である。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 法の適用例として、自動車の形状を解像する様子を図 2 に示す。等間隔直交格子では、図 2 (a) のように計算領域全体を高解像度にするため、不要に多数の格子点数が使用されている。それに対し、AMR 法では、図 2 (b) のように自動車近傍に細かい格子が集められており、全体の格子点数は必要最小限に抑えられている。

AMR 法を導入したステンスル計算の大規模計算にはいくつか先行事例があり、たとえば、マントル対流の計算 [4] や Navier-Stokes 方程式に基づく大規模流体計算 [5] が挙げられる。しかしながら、これらの先行事例はいずれも CPU による計算が主で、GPU 計算においては十分な性能が得られていない。AMR 法は、等間隔直交格子に比べて格子点の並びがリーフ単位で不規則になり GPU のメモリアクセスの効率が悪いこと、規則的なステンスル計算の実行できる最小単位（リーフ）が小さく GPU で十分な並列数が確保できないこと、オンボードメモリ上に物理変数を置く必要がありメモリ階層が増えることなど、AMR 法には GPU 計算に不利な要件が数多く存在する。AMR 法の GPU 計算において高い実行性能を得るためには、GPU 計算に適した AMR 法のデータ構造や高速化手法について検討する必要がある。

AMR 法を導入したステンスル計算の GPU 計算による大規模化と高効率化は重要なテーマであり、現在、3 次元 27 点ステンスル計算である D3Q27 格子ボルツマン法 (Lattice Boltzmann Method; LBM) に AMR 法を導入して GPU 計算に実装を進めている。本稿では、AMR 法の GPU 計算で用いるデータ構造と計算の最適化手法について報告する。AMR 法で複数の解像度の格子を設定した場合に必要な追加の処理、ベンチマーク問題による精度検証などについては他の研究報告 [6] に譲り、本稿では、最も基礎的な部分として、リーフ内部のステンスル計算の高速化について述べる。

2. 計算手法

2.1 適合格子細分化法 (AMR 法)

AMR 法の構造には、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。AMR 法の木データ構造とそれに対応する細分化格子の例を図 3 に示す。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 の末端のリーフには一定数の点を持つ格子が割り当てられる。リーフの格子は、隣接点の解像度が異なる場合に行う空間・時間補間の計算精度を確保するため、 x, y, z 軸方向にそれぞれ 2 の累乗+1 の数の格子点を持つ等間隔

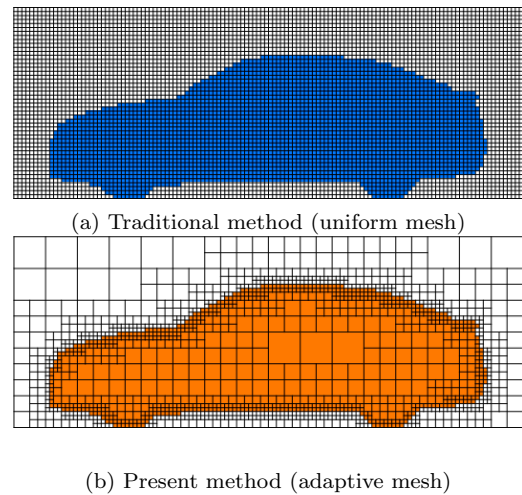


図 2 適合格子細分化法 (AMR 法) の適用例
Fig. 2 Example of Adaptive Mesh Refinement (AMR)

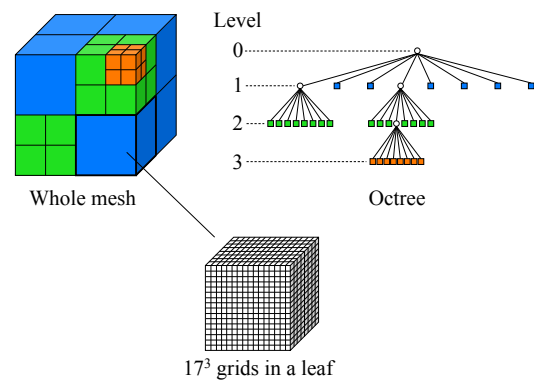


図 3 木データ構造 (octree) に基づく細分化格子
Fig. 3 Refined mesh based on octree

直交格子にすることが望ましい。また、解像度の適合性の観点から、リーフの格子点数は大きすぎないことが求められる。このため本稿では、リーフに 17^3 の数の格子点を割り当てる。

Octree の葉ノードの深さを、細分化レベル、または単にレベルと称し、リーフの格子の細かさを表す指標として用いる。レベルが 1 大きくなるごとにリーフの解像度は 2 倍になる。図 3 の例では、最も粗い青色リーフのレベルは 1、最も細かい橙色リーフのレベルは 3 であり、全体で 4 倍の解像度の差がある。

2.2 格子ボルツマン法 (Lattice Boltzmann Method; LBM)

格子ボルツマン法 (LBM) は主に非圧縮性の流体现象をシミュレートする数値計算手法の一種である。LBM は、並進・衝突と呼ばれる 2 種類の計算から構成される単純かつ並列性の高い計算アルゴリズムを有するため、GPU 計算に適した手法として実用的な大規模流体計算 [1] [7] やステンスル計算のベンチマークテスト [8] などに幅広く応用さ

れている。

LBM は気体分子運動論に基づく計算手法である。連続体で記述される流体を、有限個に離散化された速度を持つ仮想粒子の集合体（速度分布関数）と仮定し、仮想粒子が相互作用しながら空間中を移動する過程を逐次計算することで流体現象を解く。流体の密度や流速などの巨視的量は、同一格子点上に存在する仮想粒子のモーメントを取ることで得られる。仮想粒子の速度の離散化にはいくつか種類があるが、本稿では、複雑境界条件にも適用できるモデルとして、3次元で速度が27個の方向に離散化されるD3Q27 LBMを対象とする。

仮想粒子の速度は、仮想粒子が1タイムステップでちょうど隣接格子に移動する方向と大きさに限定して離散化されるため、仮想粒子の移動を計算する並進過程は図1(c)のような単純な27点ステンシル計算となる。また、仮想粒子の相互作用を計算する衝突過程は、同一格子点上の27個の速度分布関数の相互作用であり、非常に単純な同一格子点上の計算となっている。並進・衝突による1格子点の計算は、次のような順序で行う。

- 1 並進過程：27個それぞれの速度分布関数を隣接点のメモリからレジスタに読み込む
- 2 衝突過程：レジスタに読み込んだ27個の値を用いて速度分布関数の相互作用を計算する
- 3 計算結果をレジスタからメモリに書き込む

ここで、並進・衝突の2つの計算を同一カーネルで実行することで、メモリアクセスの回数を減らし、実行性能の向上を図っている。すべての計算格子点に対して上記のような計算を施すことで、1タイムステップ分の計算が完了する。

27個のそれぞれの速度分布関数は、隣接点のうちのいずれか1点からしか参照されず、1タイムステップ中に他で再利用されない。これは、隣接点から同じ値が数回参照される拡散方程式などとは大きく異なる特徴である。計算の実装の際には、それぞれの速度分布関数の参照される方向を意識したチューニングを行う必要がある。

3. AMR法のGPU実装における高速化手法

AMR法の格子点は、構造格子に比べてメモリアドレスが不連続であるため、アクセスパターンが悪く、ステンシル計算の実行性能が低くなり易い。AMR法を導入した格子ボルツマン法の計算は、Naïveな実装に対して計算時間を測定したところ、格子点数を同一にした等間隔直交格子に比べて計算時間が50倍以上も掛かっていた。AMR法によって計算を効率化するためには、計算時間の増大率を格子点数の削減率に比べて小さくしなければならず、計算の高速化は必須である。

本章では、AMR法を導入した格子ボルツマン法のGPU実装におけるステンシル計算の高速化手法について述べる。

3.1 内部格子と外殻格子のカーネル分離

AMR法において、細分化格子の最小単位であるリーフは複数の格子点から構成されており、リーフの内側（内部格子）と境界上（外殻格子）でステンシル計算における隣接点参照のアクセスパターンが異なる。すなわち、内部格子は等間隔直交格子と同様に単純なインデックス計算によって隣接点を参照することができる一方で、外殻格子は、隣接点の一部が同一リーフ内ではなく隣接リーフ上に存在しているため、隣接リーフへのアクセスを必要とする。

外殻格子の計算において隣接リーフの格子点にアクセスするためには、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、メモリアクセス回数が多くなる。また、隣接格子点のメモリアドレスが不連続になるため、アクセスパターンも悪くなる。

外殻格子のステンシル計算は、このようなメモリアクセスパターンの悪さから、内部格子よりも多くの計算時間を要する。内部格子と外殻格子を同時に計算した場合には、Warp divergenceが発生し、内部格子の計算に外殻格子と同じくらいの時間が掛かるようになり、実行性能が著しく低下する。

内部格子と外殻格子を別のカーネルで計算するようにし、内部格子の計算時に不要な条件分岐判定や、内部格子と外殻格子の計算の間に発生するWarp divergenceを排除した。この技法は、一般にBoundary peelingと呼ばれ、境界条件カーネルの分離や、メモリ分散型並列計算におけるプロセス間通信の分離など、等間隔直交格子の計算においても広く用いられている。

3.2 内部格子の高速化

内部格子の計算は等間隔直交格子の計算に似て、アクセスパターンが良く、実行性能を出しやすい。しかしながら、リーフあたりの格子点数が 17^3 と非常に小さいため、 129^3 や 257^3 のように大きな等間隔直交格子に比べると、GPU計算で性能を出すのはそれほど容易ではない。本節では、内部格子のメモリアクセスを限界まで効率化し、小さいリーフにおいても大きい等間隔直交格子に匹敵する実行性能を得るための高速化手法について述べる。

GPUにおけるメモリアクセスにはコアレッシングにより、適切にアライメントされた32, 64または128 ByteのブロックにHalf Warp（Warp内の前半または後半の16スレッド）が一斉にアクセスするような場合に最大の実効メモリバンド幅が得られる。たとえば、Half Warpが単精度浮動小数点数を読み込む場合、16個のデータがメモリ上で連続しており、かつ、メモリのブロックの先頭アドレス値が128, 192など64の倍数となっていることが求められる。

リーフの格子は 17^3 の点が規則正しく並べられているが、1列を64の倍数のByte数で取ることが出来ず、コアレッシング条件を満たしていない。そこで、リーフの格子

のメモリアドレスを GPU が効率的にアクセスできるように並べ替える。リーフの格子を $16 \times 17 \times 17$ と $1 \times 17 \times 17$ の部分に分け、 $1 \times 17 \times 17$ の部分には、それぞれの速度分布関数の参照方向に応じて $i = 0$ または $i = m - 1$ の部分を取る。すなわち、 x 軸の負方向の隣接点を参照する速度分布関数では $i = m - 1$ の側を、 x 軸の正方向の隣接点を参照する速度分布関数では $i = 0$ の側を $1 \times 17 \times 17$ に取る。これにより、読み込み・書き込みとも効率よくアクセスすることができる。また、Half Warp は内部格子の 1 列分 (15 格子点) だけを計算し、1 つのスレッドで計算を放棄する。これにより、コアレスニング条件が大きく改善され、計算が高速化されることが期待できる。

Half Warp で計算を放棄していた 1 スレッドも活用して完全なコアレスニングを実現する。まず、Half Warp 内の 16 スレッドで、 $16 \times 17 \times 17$ の部分の格子から、読み込むべき 15 個の隣接格子点を含む 1 列 16 個の格子点をコアレスニングで読み込む。次に、Warp Shuffle により 16 個読み込んだ格子点のうち必要な 15 個を、必要なスレッドに転送する。ここで、Warp Shuffle とは、同一 Warp 内のスレッド同士でレジスタの値を共有する関数で、1 クロックで値の転送が完了するため、Shared Memory を使うよりも高速にスレッド間で値の転送が出来る。次に、27 個の速度分布関数の相互作用 (衝突過程) を計算する。最後に、計算結果をメモリに書き込む際には、読み込み時と逆の方向に Warp Shuffle を行い、完全コアレスニングなメモリ書き込みを行う。このとき、Half Warp のうち 1 スレッドは外殻格子のメモリ位置に無意味なデータを書き込むことになるが、コアレスニング条件のため、無駄な計算と書き込みを行った方が Time-To-Solution の実行性能が向上する。また、外殻格子に書き込まれた無意味なデータは外殻格子の計算で上書きされるため、計算結果にも影響はない。

3.3 外殻格子の高速化

(1) 重合マザーリーフ法による外殻格子点数の削減

3.1 節で説明したように、リーフの外殻格子は隣接点参照のメモリアccessパターンが悪く、高い実行性能が出ない。そのため、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。外殻格子の数を削減する最も簡単な方法は、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ある。しかし、実行性能は高くなるが細分化の緻密さが低下するため、単純にリーフを大きくすることは、細かい形状に適合する格子を生成するという AMR 法の本来の目的と矛盾する。

AMR 法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、重合マザーリーフ法を提案する。AMR 法の元のデータ構造と重合マザーリーフ法におけるデータ構造の差異を図 4 に示す。重合マザーリーフ法では、リーフが持つべき格子を、兄弟リーフで纏めて、8 つのリーフの

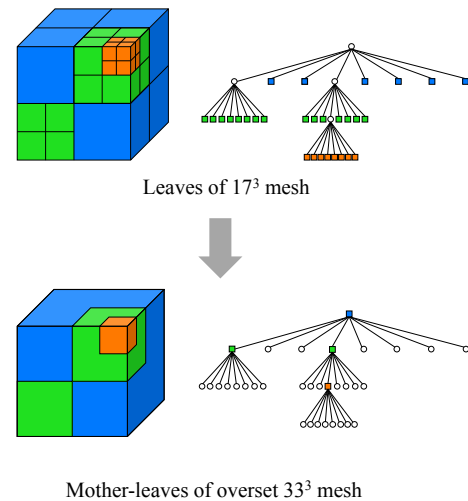


図 4 重合マザーリーフ法
Fig. 4 Overlap mother-leaf method

共通の親 (マザーリーフ) が持つ。兄弟ノードの中にリーフとリーフでないノードが混じっている場合には、リーフでないノードにも仮の格子を割り当て、マザーリーフで纏めて管理する。このため、格子の細かさが変化する境界の付近では、複数の解像度の格子が重なり合って存在している。粗い格子の値を重なり合う細かい格子の値で上書きすることで、元の AMR 法と同一の計算結果が得られる。

重合マザーリーフ法によって、見かけ上の格子の構造を変化させることなく外殻格子の削減をすることができる。マザーリーフの格子点数は、リーフの 17^3 に対して 33^3 となるため、同じサイズの問題に対して外殻格子の数は最大で 50%削減され、計算の高速化において非常に有効である。なお、3.2 節で行ったメモリアドレスの並び替えは、マザーリーフの格子点数に合わせて、 $32 \times 33 \times 33 + 1 \times 33 \times 33$ という順序に変更する。この場合、Half Warp ではなく 1 Warp で 1 列の格子点の計算を行うことで、Warp Shuffle によって完全コアレスニングなメモリアccessを行うことが出来る。

(2) C++テンプレートによる外殻格子のカーネル分離

外殻格子は内部格子とは異なり、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。元々はそのアクセスパターンをカーネル内の条件分岐文によって判定していたが、多数の条件分岐があったため、条件分岐の評価に多大な計算時間が費やされていた。それに加えて Warp divergence も発生しており、実行性能は著しく低下していた。

存在しうるアクセスパターンをすべて列挙し、別々のカーネル関数に分離して記述する方法を提案する。27 点ステンシル計算においては、外殻格子の位置のパターンが 26 通りあることに加え、そのそれぞれに隣接点参照が 27 通りあるため、それらの組み合わせによりアクセスパター

ンの数は 702 にも及ぶ。これほど多数のパターンを手動で列挙することは、プログラムの生産性を考えると現実的でない。プログラムの生産性を損なうことなく、多数の条件分岐や Warp divergence による性能低下の問題を解決するため、C++テンプレートにより多数の GPU カーネルを自動生成する。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。図 5 に、リーフ内の格子点を隣接点へのアクセスパターンごとに分離した図を示す。27 点ステンシル計算の場合、隣接点参照のアクセスパターンは、以下の 27 パターンの格子に分けられる。

$$\begin{aligned} \text{内部格子}(i, j, k) &\in \{1, 2, \dots, m-2\}^3, \\ \text{面の格子}(i, j, k) &\in \{0, m-1\} \times \{1, 2, \dots, m-2\}^2, \\ \text{辺の格子}(i, j, k) &\in \{0, m-1\}^2 \times \{1, 2, \dots, m-2\}, \\ \text{頂点格子}(i, j, k) &\in \{0, m-1\}^3 \end{aligned}$$

ここで、 m^3 はリーフの格子点数で、 $(i, j, k) \in \{0, 1, 2, \dots, m-1\}^3$ はリーフ中の格子のインデックスである。外殻格子は、6 種類の面、12 種類の辺、8 種類の頂点に分けられる。面の格子はただ 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。頂点格子は 7 つの隣接リーフにアクセスする。

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 を次のように定める。

$$\begin{aligned} gi &= \begin{cases} -1 & (i = 0) \\ 0 & (i = 1, 2, \dots, m-2) \\ 1 & (i = m-1) \end{cases} \\ gj &= \begin{cases} -1 & (j = 0) \\ 0 & (j = 1, 2, \dots, m-2) \\ 1 & (j = m-1) \end{cases} \\ gk &= \begin{cases} -1 & (k = 0) \\ 0 & (k = 1, 2, \dots, m-2) \\ 1 & (k = m-1) \end{cases} \end{aligned}$$

gi, gj, gk をテンプレート引数とし、Program 1 のようなカーネル関数を記述する。各スレッドの担当するリーフ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) の組み合わせによって定める。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、 $gi, gj, gk$  を変化させながら呼び出すことで、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カーネル関数が生成される。なお、leaf_bound<0,0,0>は内部格子に対応しており、3.2 で別途最適化されたカーネルを用いるため、ここでは使用し
```

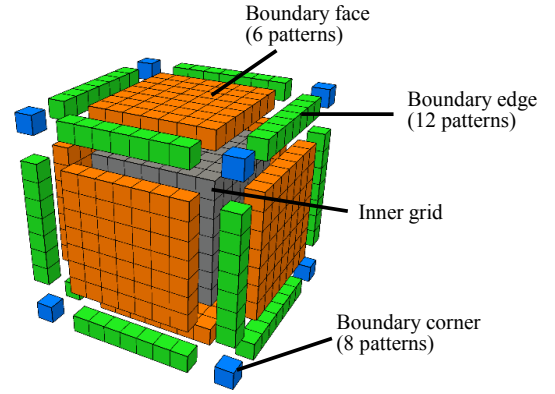


図 5 隣接点参照パターンに基づく格子の分類
Fig. 5 Mesh defusing based on stencil access patterns

表 1 計算条件

Table 1 Computational condition

mesh size per leaf	4913 (17^3)
number of leaves	512
number of whole grids	2,515,456
type of floating point number	single precision

ない。

`read_stencil` は隣接点の値を参照するサブルーチンである。 gi, gj, gk に加えてステンシルの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。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。この if 文は、 gi, gj, gk, si, sj, sk がテンプレート引数であるためコンパイル時に評価され、実行時には条件分岐判定の処理は行われない。多数の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を無くすることができ、高速な計算が可能となる。

カーネル関数を多数に分離したことで、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いる。1 つのカーネルでは GPU 計算に十分な並列数を取ることができないため、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している。

4. ステンシル計算の性能評価

AMR 法を適用した LBM に 3 章で説明した高速化手法を適用し、性能評価を行った。GPU には NVIDIA Tesla K20Xm を用い、CUDA のバージョンは 6.5 とした。計算条件を表 1 に示す。なお、空間補間や境界条件などの特殊な処理を含まない純粋なステンシル計算の部分の性能を測定するため、全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。チューニングを行っていない Naïve な実装から、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し、

Program 1 C++ template kernel for boundary grids

```
template<int gi, int gj, int gk>
__global__ void leaf_bound(...) {
    /* define distribution function of LBM */
    float f[27];
    /* calculate leaf number l and mesh index i,j,k from gi,gj,gk and thread id */
    const int l = ...;
    const int i = ..., j = ..., k = ...;
    /* evaluate streaming step */
    f[0] = read_stencil<gi, gj, gk, 0, 0, -1>(l, i, j, k, ...);
    f[1] = read_stencil<gi, gj, gk, 0, 0, 0>(l, i, j, k, ...);
    f[2] = read_stencil<gi, gj, gk, 0, 0, 1>(l, i, j, k, ...);
    ... /* more 24 "read_stencil" for remained stencil patterns */
    /* calculate collision step */
    f[0] = ...;
    f[1] = ...;
    /* write f[] to global memory */
    .....
}
```

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, ...) {
    float retval; //return value
    if(gi * si > 0 || gj * sj > 0 || gk * sk > 0) {
        retval = ...; /* read value from neighbor leaf */
    } else {
        retval = ...; /* read value from this leaf (i + si, j + sj, k + sk) */
    }
    return retval;
}
```

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（内部＋外殻）のそれぞれについて測定した。

計算時間の測定結果を表 2 に示す。表 2 において、Inner は内部格子の計算時間、Boundary は外殻格子の計算時間であり、Total は全格子の計算時間を表している。Naïve は高速化手法を適用していないときの計算時間であり、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、全格子の計算時間のみ示している。表中の「3.1」、「3.2」などは適用した高速化手法を表しており、3 章で述べた手法の説明の節番号などに対応している。

内部格子と外殻格子のカーネル分離 (3.1) によって 4.35 倍の高速化に成功している。外殻格子の計算に内部格子の 5 倍程度の時間が掛かっており、外殻格子の計算が非常に低速であることがわかる。

内部格子の計算カーネルの高速化 (3.2) により、内部格子の計算が 1.95 倍、外殻格子の計算が 1.16 倍に高速化され、全体としては 1.24 倍の高速化を達成した。内部格子だけでなく外殻格子の計算まで高速化されたのは、格子点の

表 2 それぞれの高速化手法の適用前後の計算時間（ミリ秒）

Table 2 Computation time on each tuning applied [msec]

	Inner	Boundary	Total
Naïve	—	—	142.187
3.1	5.166	27.506	32.672
3.2	2.642	23.731	26.373
3.3 (1)	2.705	12.111	14.816
3.3 (2)	2.705	5.270	7.975

位置とメモリアドレスの対応を並べ替えたことにより外殻格子の計算でもメモリアクセスのコアキャッシングが改善されたためと考えられる。

重合マザーリーフ法 (3.3 (1)) により、外殻格子の格子点数が削減され、外殻格子の計算は 1.96 倍に高速化された。外殻格子とは逆に内部格子では、格子点数が増加したため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効果が大きく、全体としては 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離 (3.3 (2)) では、外殻格子の計算が 2.29 倍高速化された。条件分岐文の排

除によって、条件分岐判定のコストや Warp divergence がなくなったこと、ローカルメモリに配置されていたローカル変数をレジスタに配置できるようになったことなどが要因として考えられる。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、チューニングを一切行っていないプログラムから合計で 17.8 倍の高速化が達成された。特に高速化の効果が高かった手法は、2 度の計算カーネルの分離 (3.1 および 3.3 (2)) で、それぞれ 4.35 倍および 1.86 倍の高速化が達成されている。AMR 法を導入したステンシル計算には多数のアクセスパターンが存在しており、それらが計算の実行性能に強く影響しているということがいえる。

5. おわりに

本研究では、ステンシル計算に適合細分化格子法 (AMR 法) を導入して GPU 計算に実装するためのデータ構造やアルゴリズムを提案し、その際に適用した種々の高速化手法について述べ、それぞれの高速化手法の効果を確かめた。AMR 法におけるステンシル計算の隣接点参照は格子点の位置によってメモリアccessパターンが異なるため、隣接点参照のアクセスパターンに応じて格子点を分類し、それぞれを異なるカーネルで計算することが望ましい。C++ テンプレートをを用いて 1 種類のみ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た複数の GPU カーネル関数を生成し、コンパイル時にアクセスパターンの判定をすることで、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。今回実装したのは 27 点ステンシル計算の D3Q27 LBM であるが、他の種類のステンシル計算においても C++ テンプレートをを用いて GPU カーネル関数を生成する技法は有効であり、その汎用性は高い。

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」から支援を頂いた。記して謝意を表す。

参考文献

- [1] 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ム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).
- [2] 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, *Proceedings of the 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).

- [3] Rossinelli, D., Hejzali Hosseini, 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, *Proceedings of the 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).
- [4] Burstedde, C., Ghattas, O., Gurnis, M., Isaac, T., Stadler, G., Warburton, T. and Wilcox, L.: Extreme-Scale AMR, *Proceedings of the 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).
- [5] Malhotra, D., Gholami, A. and Biros, G.: A Volume Integral Equation Stokes Solver for Problems with Variable Coefficients, *Proceedings of the 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).
- [6] 長谷川雄太, 青木尊之, 小野寺直幸: 格子細分化を導入した D3Q27 格子ボルツマン法の GPU 実装, 第 28 回数値流体力学シンポジウム講演論文集, Vol. 28, No. E08-3 (2014).
- [7] 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, *Proceedings of the 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).
- [8] Nguyen, A., Satish, N., Chhugani, J., Kim, C. and Dubey, P.: 3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs, *Proceedings of the 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.2010.2 (2010).