Vol.2014-ARC-213 No.25
Vol.2014-HPC-147 No.25
2014/12/10

ステンシル計算のための GPU コンピューティング・フレームワークのチューニング高度化

下川辺 隆史[1,a)]　青木 尊之[1]　小野寺 直幸[1]

概要：格子に基づいたシミュレーションは高性能計算分野において重要なアプリケーションである。本研究では、GPU による格子に基づいたシミュレーションを簡便に記述し、高い生産性で開発することを可能とする GPU コンピューティング・フレームワークを開発する。フレームワークは C++言語のテンプレートで実装され、高い可搬性を実現している。GPU による格子計算では、その性能は計算に用いるスレッド数などの実行時パラメータに大きく依存する。実行時に最適なパラメータを自動選択する機構をフレームワークに導入し、その性能について評価する。

1. はじめに

ここ数年、高い浮動小数点演算能力と高いメモリバンド幅および電力効率のよい Graphics Processing Units (GPU) が汎用計算に多く用いられるようになってきた。東京工業大学の GPU スパコン TSUBAME2.5 に搭載された NVIDIA Tesla K20X GPU は 1.31 TFlops の性能を持ち、250 GB/s のバンド幅に達している。GPU により様々な格子に基づく物理アプリケーションが大幅に性能向上したことが多く報告されている [1], [2], [3], [4].

多くの高速化の報告があり，GPU 計算は高い性能が得られることが期待されるものの，GPU 用プログラミングは、NVIDIA 社製 GPU に特化した CUDA[5] や複数のマルチコアプロセッサに対応した言語 OpenCL [6] などを用いる必要があり、さらにプロセッサの性能を引き出すためには個々のアーキテクチャを意識したプログラミングを行い、機種固有の最適化手法を導入する必要がある。このような問題を解決するために、高い抽象度により生産性を向上させ、可搬性を備えたいくつかのプログラミングモデルが提案されている。格子計算においては，異種混合型スパコンで利用できるドメイン特化型言語（Domain specificlanguage; DSL）の Physis [7] や CUDA GPU に対応した Mint [8] などが提案されている.

直交格子上で実行される数値計算を高生産に GPU スパコン上に実装するための GPU コンピューティング・フレームワークを開発を進めている [9], [10]。このフレームワークは、C/C++言語および CUDA を用い実装され、大規模な格子計算で必要なステンシル計算を簡便に表現するクラスや GPU 間通信やノード間通信を簡単に行うクラスを提供する。これにより、高い生産性を実現し、性能を出すためには制約の多い GPU アーキテクチャや GPU 間通信の実装を意識することなく、GPU スパコン向けの最適化を施すことが可能である。ユーザは、ステンシル計算のみを記述するため、可搬性があり、本フレームワークを用いることで GPU 以外のアーキテクチャでもユーザコードの変更無く実行することができるようになる。

本研究では、開発を進めている GPU コンピューティング・フレームワークに、実行時に最適なパラメータを自動選択する機構を導入する。GPU によるステンシル計算では、その性能は計算に用いるスレッド数などの実行時パラメータに大きく依存する。実アプリケーションでは、様々な計算格子サイズに対して計算を行うが、その計算毎に計算パラメータを最適化することは難しい。本フレームワークでは、ユーザはステンシル計算のみ記述するため、アプリケーションコードを変更することなく、その実行時パラメータを調整することが可能である。本論文では、フレームワークが提供するステンシル計算環境を拡張し、実行時に最適なパラメータを自動選択する機構を導入したので、その実装、プログラミングモデルと、それを拡散方程式に適用した時の実行性能について述べる。

2. GPU コンピューティング・フレームワーク

ここでは、開発を進めている GPU コンピューティング・フレームワークについて述べる。本フレームワークは、複数 GPU 計算に対応しているが、本研究は単一 GPU 計算

[1] 東京工業大学
Meguro, Tokyo, 152–8550, Japan
[a)] shimokawabe@sim.gsic.titech.ac.jp

Vol.2014-ARC-213 No.25
Vol.2014-HPC-147 No.25
2014/12/10

を対象としているため、単一 GPU 計算に関する事項のみ記述する。なお、詳細は参考文献 [9], [10] を参照されたい。

本フレームワークは、直交格子型の解析を対象とし、各格子点上で定義される物理変数（プログラム上は配列となる）の時間変化を計算する。また、当該物理変数の時間ステップ更新は陽的であり、ステンシル計算によって行われる。TSUBAME に導入されている NVIDIA 社製 GPU で実行することを目指し、実装には、ホストコードは C/C++ 言語、デバイスコードは CUDA を用いる。

フレームワーク設計における主な目標を述べる。

- プログラマは格子点上での計算についてのみ記述する。格子全体の処理はフレームワークが行う。格子全体の処理がユーザコードからフレームワークへ分離され GPU に合った最適化手法を隠蔽する。また、バックエンドとして様々なプロセッサを採用することができ、拡張性と高い生産性を持つ。現在、フレームワークでは、GPU および CPU で実行可能である。
- フレームワークが提供するテンプレートを用いた C++ クラスを用い格子点上の計算を記述する。言語拡張や標準的でないプログラミングモデルを利用すると、既存コードからの乖離も大きく利用しにくい。また、拡張部分がフレームワークを他の環境へ移植する妨げになりうる。その点、C++のテンプレートやクラスは広く使われており、基盤とできる。

以上まとめると、ユーザは、(1) フレームワークの提供するテンプレート関数およびクラスを用い、ステンシル計算を行う関数を記述する。(2) フレームワークの提供するクラスを用いて、GPU 間通信を記述する。以下で詳細について記述する。

2.1 ステンシル計算関数の定義

本フレームワークでは、ステンシル計算は、フレームワークの提供する ArrayIndex3D 等を用い、ファンクタ（関数オブジェクト）として定義し、ステンシル計算関数と呼ぶ。3 次元の拡散計算では、次のように関数を定義できる。

```
struct Diffusion3d {
__host__ __device__
void operator()(const ArrayIndex3D &idx,
  float ce, float cw, float cn, float cs,
  float ct, float cb, float cc,
  const float *f, float *fn) {
  fn[idx.ix()] = cc*f[idx.ix()]
  +ce*f[idx.ix<1,0,0>()]+cw*f[idx.ix<-1,0,0>()]
  +cn*f[idx.ix<0,1,0>()]+cs*f[idx.ix<0,-1,0>()]
  +ct*f[idx.ix<0,0,1>()]+cb*f[idx.ix<0,0,-1>()];
}
};
```

ArrayIndex3D は、対象とする配列のサイズ (n_x, n_y, n_z) を保持し、ある特定の格子点を表すインデックス (i, j, k) を設定できる。対象とする配列が f であるとき、idx を ArrayIndex3D のオブジェクトとすると f[idx.ix()] として使われ、これは配列 f の (i, j, k) 点の値を返す。ArrayIndex3D はテンプレートを用いたメンバ関数が定義されており、例えば、idx.ix<+1,0,0>()、idx.ix<-1,-2,0>() とすると、$(i+1, j, k)$、$(i-1, j-2, k)$ を表すインデックスを返す。テンプレートを用いることで、GPU および CPU でのインデックス計算の高速化を図っている。

ステンシル計算関数の第一引数は固定で、計算対象となる格子のインデックス情報を持つ idx を受け取らなければならない。関数実行時には、格子点 (i, j, k) の値が設定されているため、(i, j, k) を中心としたステンシル計算を関数内に記述する。f, fn は配列へのポインタであり、これに対しステンシルアクセスすることとなる。

拡散係数が空間の関数になっているなど、解析する問題によっては f, fn 以外の係数を保持する変数が必要となる。ステンシル計算関数内では、ある格子点を更新するための記述しか表現できないため、空間の関数になっている係数に対しては f, fn と同様に配列として確保し、f, fn と同じようにステンシル計算関数の外から渡す必要がある。

2.2 ステンシル計算関数の実行

本フレームワークは、全ての格子点に計算を行う Loop3D 等のクラスを提供する。これを用い、ステンシル計算関数の実行は以下のように行う。

```
Loop3D loop3d(nx+2*mgnx, mgnx, mgnx,
              ny+2*mgny, mgny, mgny,
              nz+2*mgnz, mgnz, mgnz);
loop3d.run(Diffusion3d(), ce, cw, cn, cs,
           ct, cb, cc, f, fn);
```

Loop3D はステンシル関数を適用する範囲を指定するパラメータで初期化する。Loop3D::run() は任意個の異なる型を引数にとるテンプレート関数として定義されている。C++のテンプレート関数の型推論を利用し、Loop3D::run() は、与えられた全ての引数を第二引数以降に持つファンクタ Diffusion3d() を呼び出す。ファンクタは、NVIDIA CUDA の__host__、__device__で定義することができ、ホスト、デバイス両方へコンパイルされており、Loop3D::run() が CPU 上のデータに対しても、GPU 上のデータに対しても同じ関数を実行する。Loop3D 内部の実装としては、CPU 上で実行する場合はファンクタを全格子点に対して for 文で実行し、GPU 上で実行する場合は内部で生成される CUDA のグローバル関数に包み、適当な CUDA のブロック数、スレッド数が渡され全格子点に対して実行される。第二引数以降で初めて出てくるポインタが GPU メモリの配列を指すか CPU メモリの配列を指すかを判定し、GPU で実行するか CPU で実行するかを

Vol.2014-ARC-213 No.25
Vol.2014-HPC-147 No.25
2014/12/10

自動的に決定する。

3. フレームワークの高度化

ステンシル計算関数は、これまで GPU および CPU で実行可能であったが、その実行方法についてユーザは指定することはできない。本研究では、本フレームワークが提供する Loop3D 等を拡張し、ユーザ定義の実行方法を指定できるように拡張する。この拡張された機能を用いて、自動チューニング機能を付加した GPU 計算方法を実装し、その実行性能を評価する。

3.1 ステンシル計算関数の実行環境の拡張

本フレームワークが提供する Loop3D を C++のテンプレートパラメータを引数に取るクラスへ拡張し、全ての格子点に対して実行する計算方法を変更できるようにする。拡張された Loop3D を用いると、ステンシル計算関数は下記のように実行できる。

```
Loop3D<HostLoopEngine3D>
      loop3d(nx+2*mgnx, mgnx, mgnx,
             ny+2*mgny, mgny, mgny,
             nz+2*mgnz, mgnz, mgnz);
loop3d.run(Diffusion3d(), ce, cw, cn, cs,
           ct, cb, cc, f, fn);
```

HostLoopEngine3D が新しく導入されたテンプレートパラメータの引数として与えられたもので、これは CPU 上で全格子点に対してファンクタとして指定されているステンシル関数を for 文で実行する。パラメータとして DeviceLoopEngine3D を指定することで GPU 上で実行することが可能である。クラスの初期化や Loop3D::run() で指定されるパラメータは従来と同等である。このテンプレートパラメータには、ユーザ定義型を指定することが可能となっており、ユーザによって計算方法を変更することが可能となっている。

テンプレートパラメータとして指定できるクラスの例として、HostLoopEngine2D を以下に示す。

```
class HostLoopEngine2D {
public:
HostLoopEngine2D(const LoopRange r[2])
    : r_(r) {}
template <class Func, class Params>
void operator()(Func func, const Params &p) {
  ArrayIndex2D idx(r_[0].n, r_[1].n);

  for (int j=r_[1].begin; j<r_[1].end; j++) {
  for (int i=r_[0].begin; i<r_[0].end; i++) {
    idx.set_pos(i, j, k);
    run_func_host(idx, func, params);
  }
  }
}
private:
  const LoopRange *r_;
};
```

HostLoopEngine2D は、ステンシル関数を適用する x および y 方向の範囲を受け取り、初期化される。ユーザはこれを用いて、operator() 内でステンシル関数を全格子点上で実行する。LoopRange::n、LoopRange::begin、LoopRange::end はそれぞれ適用範囲の長さ、適用範囲の始点、適用範囲の終点を表す。ArrayIndex2D は、ArrayIndex3D の 2 次元版で、x および y 方向の範囲の長さで初期化され、ArrayIndex3D::set_pos() で計算する格子点の位置を指定する。Func および Params には、それぞれユーザ指定のステンシル計算関数およびそれに与えるパラメータが格納されている。Params をテンプレートパラメータとして指定することで、ステンシル計算関数が受け取る任意の型の任意の個数の変数をもつ構造体を扱うことが可能となっている。なお、現在の実装では、拡張された Loop3D ではリードオンリキャッシュを有効にすることができない。これは、記述性を向上させるため上記の Params でステンシル関数に渡すパラメータを受け渡ししており、これによってキャッシュがうまく動作しないものと思われる。

3.2 最適パラメータ選択機能付きステンシル計算関数の実行環境

本フレームワークでは、DeviceLoopEngine3D を拡張したクラスとして、自動チューニング機能を付加した DeviceAutoTuningLoopEngine3D を提供する。ステンシル計算関数を GPU 上で実行するには、適当な CUDA ブロック数、スレッド数を指定する必要がある。しかし、計算格子サイズによって最適なこれらの数は異なる。そこで、DeviceAutoTuningLoopEngine3D は最適なスレッド数でステンシル関数を実行することを可能とする。

DeviceAutoTuningLoopEngine3D をテンプレートパラメータに指定する時、ステンシル計算は次のように記述する。

```
DeviceAutoTuningInfo info;
Loop3D<DeviceAutoTuningLoopEngine3D>
      loop3d(nx+2*mgnx, mgnx, mgnx,
             ny+2*mgny, mgny, mgny,
             nz+2*mgnz, mgnz, mgnz);
loop3d.run(Diffusion3d(), &info, ce, cw, cn, cs,
                          ct, cb, cc, f, fn);
```

Loop3D の初期化はこれまでと同じで、Loop3D::run() は従来どおり任意個の異なる型を引数にとるが、第二引数に DeviceAutoTuningInfo のオブジェクトへのポインタを指定する。DeviceAutoTuningInfo は、自動チューニングするパラメータや計算条件を変更して実行したステンシル関数の実行時間を保持するクラスであり、ユーザプロ

Vol.2014-ARC-213 No.25
Vol.2014-HPC-147 No.25
2014/12/10

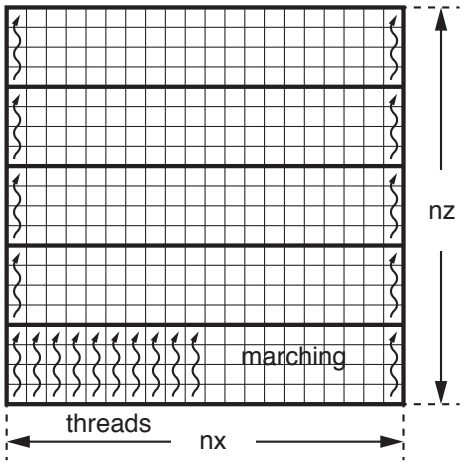

図 1 z 方向のマーチングを用いた 3 次元計算（xz 断面）
Fig. 1 GPU computation of 3D computational mesh with the z-marching (X-Z plane).

グラム中で `Loop3D::run()` 毎にそのオブジェクトを用意する必要がある。第三引数以降はこれまで通りファンクタ `Diffusion3d()` へ与えられる。

`DeviceAutoTuningLoopEngine3D` は、CUDA のブロックを二次元に確保し、xy 平面に平行に配置し、ステンシル計算関数を実行する。CUDA のブロック内の各スレッドをある (i, j) に割り当て、z 方向にマーチングしながら順に計算を行う。この計算方法の xz 断面を図 1 に示す。一般に、計算領域に割り当てるスレッド数を多くすると GPU では性能が向上しやすい。このため、1 スレッドで z 方向の全格子点を計算するのではなく、z 方向でいくつかの領域で分割して、CUDA のブロックを割り当てる。z 方向の分割数が d_z で、計算領域の格子点数を (n_x, n_y, n_z) とするとき、n_z/d_z の格子点をマーチングすることになる。

`DeviceAutoTuningInfo` は、現状では、自動チューニングするパラメータとして CUDA のブロック内の x と y 方向のスレッド数（すなわち、1 つの CUDA ブロックが計算する x と y 方向の格子数に等しい）と z 方向にマーチングする格子数を指定できる。自動チューニングによって選択されうるスレッド数とマーチングする格子数を表 1 に示す。`DeviceAutoTuningLoopEngine3D` が `Loop3D` に指定されると、ステンシル計算関数が実行される度に `DeviceAutoTuningInfo` によって新しい自動チューニング・パラメータが指定され、その実行時間が計測され、`DeviceAutoTuningInfo` のオブジェクトへ結果として保持される。`DeviceAutoTuningLoopEngine3D` は、全てのパラメータで計測が完了すると、それ以降のステンシル関数の実行では、実行時間を最短とする最適なパラメータを用いる。自動チューニングはプログラムの実行中に行われ、チューニング中は性能低下が見られるが、一般的に格子を用いた計算では、ステンシル関数は数万回を超え実行されるため、チューニング・パラメータ決定のための速度低下はアプリケーション全体の実行時間と比較すると無視することができる。

表 1 自動チューニングされるパラメータ
Table 1 Auto-tuning parameters.

Number of threads in x direction	4, 8, 16, 32, 64, 128
Number of threads in y direction	1, 2, 4, 8, 16
Number of mesh in z-marching	1, 2, 4, 8, 16

表 2 計算格子サイズによる最適な CUDA スレッド数と z 方向のマーチング格子数
Table 2 Optimal CUDA thread size and z-marching length for mesh sizes.

Mesh size	Threads	z-marching
32^3	$(32, 8, 1)$	2
64^3	$(64, 4, 1)$	2
256^3	$(128, 1, 1)$	2
512^3	$(128, 1, 1)$	2
$8 \times 512 \times 512$	$(8, 16, 1)$	8

3.3 評価実験

`DeviceAutoTuningLoopEngine3D` を用いた `Loop3D` による最適パラメータ選択機能の有用性を示すため、本フレームワークを用い拡散方程式を実装し、性能評価する。拡散方程式は流体計算等で多く用いられる方程式で，以下のように表される．

$$\frac{\partial f}{\partial t} = \kappa \nabla^2 f \tag{1}$$

ここで，f は物理変数，κ は拡散係数である．1 方向に 3 点，3 次元計算では 7 点の格子点を参照する．

性能測定は，GPU 上での最適化の効果を見るため、単一 GPU で行い、NVIDIA Tesla K20X GPU を用いる。Tesla K20X は単精度計算で 3.95 TFlops のピーク性能を持ち，250 GB/s のメモリバンド幅に達する．

計算格子サイズを 32^3 から 512^3 まで変化させる。また、計算領域のアスペクト比が大きい場合の例として $8 \times 512 \times 512$ の計算格子を用いた計算を合わせて行う。本フレームワークは、カーネル分割による通信を計算で隠蔽するオーバーラップ手法を提供する。この手法では、アスペクト比が大きい境界領域とそうでない中心領域に分割される。$8 \times 512 \times 512$ の計算格子は、アスペクト比が大きい境界領域を想定している。表 2 にそれぞれの格子サイズの場合で、自動チューニングによって選択された最適なパラメータを示す。最適な x 方向のスレッド数は、計算格子サイズの x 方向の長さに大きく依存しており、特にこのパラメータを実行時に最適化する必要がある。

こららの格子サイズを持つ計算領域で、最適パラメータ選択機能を用いた拡散計算と表 2 にあるパラメータに固定

Vol.2014-ARC-213 No.25
Vol.2014-HPC-147 No.25
2014/12/10

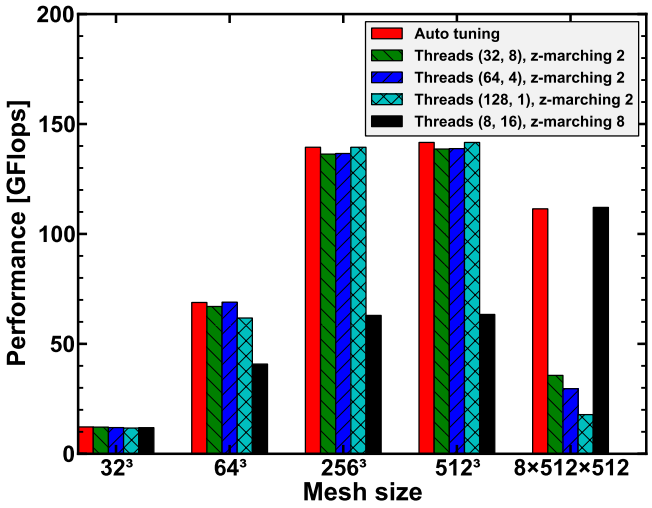

図 2 フレームワークの自動最適化を用いた拡散計算の実行性能
Fig. 2 Performance of diffusion computation with auto-auning provided by the framework.

した拡散計算を実行する。図 2 にその計算の実行性能を示す。計算は 1GPU を用い単精度で行った。図に示すように、自動最適化手法を用いた拡散計算は、全ての格子サイズの領域に対して高速な実行速度を示し、512^3 の計算格子では 141.6 GFlops を達成した。$(128, 1, 1)$ のスレッドと z 方向のマーチング格子点数に 2 を指定して実行すると多くの計算格子サイズの条件で良い実行性能が達成できるものの、$8 \times 512 \times 512$ では性能が低下し、17.8 GFlops となる。これに対して、最適パラメータ選択機能を用いた拡散計算では、スレッド数を $(8, 16, 1)$ として z 方向のマーチングする格子点数を 8 とし最適化することで、111.4 GFlops を達成した。これは、17.8 GFlops の 6.3 倍高速である。

4. おわりに

格子に基づいたシュミレーションのための GPU コンピューティング・フレームワークを開発した。フレームワークは、ステンシル計算を簡便に記述するクラスとそれを GPU および CPU で実行するクラスを提供する。これにより可搬性があり、高い生産性を実現している。本研究では、ステンシル計算関数の実行環境の拡張を行い、ステンシル計算関数を実行する方法をユーザ指定できるよう機構を導入した。この拡張された機能を用いて、自動チューニング機能を付加した GPU 計算方法を実装した。GPU 計算では、計算格子サイズによって最適な CUDA のスレッド数が異なる。自動チューニングでは、CUDA のブロック内の x と y 方向のスレッド数（すなわち、1 つの CUDA ブロックが計算する x と y 方向の格子数に等しい）と z 方向にマーチングする格子数が最適化される。評価実験では、自動最適化手法を用いた拡散算計を NVIDIA Tesla K20X GPU で実行し、パラメータを固定した場合と比較して、常に高い性能を達成することを示した。アスペクト比の高い $8 \times 512 \times 512$ の計算格子では、スレッド数を $(128, 1, 1)$ とした場合と比較して 6.3 倍の高速化を達成した。

謝辞 本研究の一部は科学研究費補助金・若手研究（B）課題番号 25870223「低消費エネルギー型 GPU ベース次世代気象計算コードの開発」，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」，学際大規模情報基盤共同利用・共同研究拠点，および，革新的ハイパフォーマンス・コンピューティング・インフラから支援を頂いた．記して謝意を表す．

参考文献

[1] 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, *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '10, New Orleans, LA, USA, IEEE Computer Society, pp. 1–11 (online), DOI: http://dx.doi.org/10.1109/SC.2010.9 (2010).

[2] Shimokawabe, T., Aoki, T., Ishida, J., Kawano, K. and Muroi, C.: 145 TFlops Performance on 3990 GPUs of TSUBAME 2.0 Supercomputer for an Operational Weather Prediction, *Procedia Computer Science*, Vol. 4, pp. 1535 – 1544 (online), DOI: DOI: 10.1016/j.procs.2011.04.166 (2011). Proceedings of the International Conference on Computational Science, ICCS 2011.

[3] Shimokawabe, T., Aoki, T., Takaki, T., Yamanaka, A., Nukada, A., Endo, T., Maruyama, N. 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, Seattle, WA, USA, ACM, pp. 1–11 (2011).

[4] 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，2013 年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 HPCS2013, Vol. 2013 (2013).

[5] NVIDIA: CUDA C Programming Guide 5.0, `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf` (2013).

[6] Khronos OpenCL Working Group: *The OpenCL Specification, version 1.0.29* (2008).

[7] Maruyama, N., Nomura, T., Sato, K. and Matsuoka, S.: Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers, *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, New York, NY, USA, ACM, pp. 11:1–11:12 (online), DOI: http://doi.acm.org/10.1145/2063384.2063398 (2011).

[8] Unat, D., Cai, X. and Baden, S. B.: Mint: realizing CUDA performance in 3D stencil methods with annotated C, *Proceedings of the international conference on Supercomputing*, ICS '11, New York, NY, USA, ACM, pp. 214–224 (online), DOI:

Vol.2014-ARC-213 No.25
Vol.2014-HPC-147 No.25
2014/12/10

http://doi.acm.org/10.1145/1995896.1995932 (2011).

[9] 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，2014 年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 HPCS2014，Vol. 2014 (2014).

[10] Shimokawabe, T., Aoki, T. and Onodera, N.: High-productivity Framework on GPU-rich Supercomputers for Operational Weather Prediction Code ASUCA, *Proceedings of the 2014 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, New Orleans, LA, USA, IEEE Computer Society, pp. 1–11 (2014).