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

下川辺 隆史[1,a)]　青木 尊之[1,b)]　小野寺 直幸[1,c)]

概要：複数 GPU による格子に基づいたシミュレーションを簡便に高い生産性で開発することを可能とするマルチ GPU コンピューティング・フレームワークを提案する．複数 GPU による格子計算では、性能を引き出すために、GPU アーキテクチャに適した計算手法を導入する必要があるが、これらは複雑で開発コストが高い．提案フレームワークは、格子上のステンシル計算を簡便に表現する C++言語のテンプレートクラスを提供する．プログラムユーザは、これを用い、格子計算のコアとなる格子点を更新する関数のみを記述し、フレームワークはこれを基に GPU 実行コードおよび CPU 実行コードを生成する．さらに、フレームワークはユーザコードを MPI と OpenMP で並列化し複数 GPU で実行し、GPU 間通信を簡単に記述するクラスを提供する．これらを用い、通常の C++ コードを記述することで、性能を出すには制約の多い 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か可能である．評価実験として、複数 GPU による拡散方程式をフレームワークを用い実装し参照実装と性能比較を行う．２基の NVIDIA Tesla K20X GPU を用いた計算ではフレームワークを用いることで 1.4 倍の高速化に成功し、複数 GPU 計算では良い弱スケーリングを示した．実アプリケーションへの適用例として、圧縮性流体計算による Rayleigh-Taylor 不安定性の成長シミュレーションをフレームワークを用い実装した．

キーワード：複数 GPU，ステンシル，格子計算，フレームワーク

A High-productivity Framework for Multi-GPU computation of Mesh-based applications

Shimokawabe Takashi[1,a)]　Aoki Takayuki[1,b)]　Onodera Naoyuki[1,c)]

Abstract: The paper proposes a high-productivity framework for multi-GPU computation of mesh-based applications. In order to achieve high performance on these applications, we have to introduce complicated optimized techniques for GPU computing, which requires relatively-high cost of implementation. Our framework automatically translates user-written functions that update a grid point and generates both GPU and CPU code. In order to execute user's code on multiple GPUs, the framework parallelizes this code by using MPI and OpenMP. The framework also provides C++ classes to write GPU-GPU communication effectively. The programmers write user's code just in the C++ language and can develop program code optimized for GPU supercomputers without introducing complicated optimizations for GPU computation and GPU-GPU communication. As an experiment evaluation, we have implemented multi-GPU computation of a diffusion equation by using this framework and achieved good weak scaling results. The framework-based diffusion computation using two NVIDIA Tesla K20X GPUs is 1.4 times faster than manual implementation code. We also show computational results of the Rayleigh-Taylor instability obtained by 3D compressible flow computation written by this framework.

Keywords: Multi-GPU, stencil, mesh-based applications, framework

1. はじめに

格子法に基づいた物理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．メモリアクセスが律速となる計算のため，これらのシミュレーションは従来のスパコンでは困難な問題であった．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) を汎用計算に用いる General-Purpose GPU (GPGPU) の研究が盛んに行われている．東京工業大学の GPU スパコン TSUBAME 2.5 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 により様々な格子法にも基づく物理アプリケーションにおいて，大幅な性能向上が可能であり，多くの報告がなされている [1], [2], [3], [4], [5], [6], [7].

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 [8] や複数のマルチコアプロセッサに対応した言語 OpenCL [9] 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 [10] や CUDA GPU に対応した Mint [11] などが提案されている．

TSUBAME2.5 などの GPU を大規模に搭載したスパコンが現れるにつれ，複数 GPU を用いた大規模な格子計算が行われている．大規模な GPU 計算では，性能を引き出すためには，GPU アーキテクチャに適した最適化手法に加え，GPU 間の効率的な通信方法を導入する必要がある．しかしながら，これらは複雑な実装となるため開発コストが非常に高い．

本論文では，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マルチ GPU コンピューティング・フレームワークを提案する．実アプリケーションの開発では，過去に作成したコードの一部を再利用したり既存のライブラリを利用したりすることが多く，マルチ GPU コンピューティング・フレームワークはこのような既存コードとの連携のしやすさは必須である．また，フレームワークに言語拡張や標準でないプログラミングモデルを導入すると，ユーザコードの可搬性や拡張性が損なわれてしまう．そこで，本論文では，従来の DSL とは異なり，独自の言語拡張や標準でないプログラミングモデルを導入することなく通常の C++コードを記述することで GPU スパコンに最適なコードを高生産に開発できるフレームワークを目指す．ユーザコードは C++言語で記述できるため可搬性と拡張性が高い．提案フレームワークは，C/C++言語および CUDA を用い実装されている．格子点上のステンシル計算を簡便に表現するクラスやノード内やノード間の GPU 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，通常の C++コードを記述することで 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．提案フレームワークでは，同一のユーザコードからコンパイル時に GPU 用コードの生成と同時に CPU 用コードを生成し，GPU と CPU の両方でユーザコードを実行することができる．

2. フレームワークの概要

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．本フレームワークは，複数のノードの搭載された複数 GPU による計算に対応する．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．

- フレームワークを用いたユーザコードの記述は C++言語を用いる．言語拡張や標準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．特に既存の外部ライブラリとの連携を考慮すると，フレームワークを用いたユーザコードが標準的言語で記述できることは重要である．また，独自の言語拡張は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．フレームワークでは変数配列に独自な型を導入せず，C/C++言語の配列とし，完全なポインタの互換性を持たせることで，ユーザコード内で自由に外部ライブラリを呼ぶことができる．
- フレームワークは，ノード内の複数 GPU 計算は単一プロセスで実行し，ノード間を MPI で並列化する．ノード内の複数 GPU 計算を単一プロセスで実行する

[1] 東京工業大学
Tokyo Institute of Technology 2-12-1-i7-3, Ohokayama, Meguro-ku, Tokyo, Japan
[a)] shimokawabe@sim.gsic.titech.ac.jp
[b)] taoki@gsic.titech.ac.jp
[c)] onodera@sim.gsic.titech.ac.jp

ことで，GPUDirect を用いた GPU 間の直接通信が可能となり，性能向上が期待できる．

- 単一プロセスが複数の GPU 計算を実行することを意識することなく，それぞれの GPU に着目してユーザコードを記述できるようにする．すなわち，ユーザサイドからは一つの GPU について時間発展計算を記述する．単一プロセスから複数 GPU を扱うにも関わらず，計算に用いる全 GPU を MPI で並列化するフラット MPI 並列とほぼ同等に記述できることとする．
- プログラマはある格子点に着目して，格子点上の物理変数の時間変化の計算を記述する．その計算を格子全体に適用する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，ユーザは通常の C++ コードを記述することで GPU 向け最適化手法を導入できる．また，分離することで格子全体に適用する処理の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，フレームワークは GPU および CPU コードを生成できる．
- 格子点上の物理変数の時間変化の計算では，フレームワークが提供する C++ クラスを用い物理変数の格納された配列へアクセスする．これにより，ユーザはある格子点に着目して，その格子点におけるステンシルアクセスのみを意識し計算を記述できる．
- 複数ノードでの複数 GPU 計算では，GPU 間通信が必須である．フレームワークは GPU 間通信を簡便に記述するクラスを提供する．これを用いることで，ユーザは通信先の GPU が同一ノード内にあるか異なるノードにあるか意識することなく，GPU 間通信を記述できる．特にプログラマはノード間通信で明示的に MPI 関数を記述する必要がなくなる．

以上まとめると，ユーザは，(1) ある GPU に着目して物理現象の時間発展計算を記述し，(2) 格子点上の物理変数の時間変化の計算では，ある格子点に着目しステンシル計算を行う関数を記述する．(3) GPU 間の通信はフレームワークの提供するクラスを用いる．以上を記述することで，ユーザコードは複数 GPU で実行することが可能となる．

3. フレームワークの実装

提案フレームワークの実装について述べる．まず，フレームワーク全体の構造について説明し，ステンシル計算関数の実行方法について説明する．次に，複数 GPU 計算で必須となる GPU 間通信の実装について述べる．

3.1 フレームワークの構造

本フレームワークは，複数 GPU 計算に対応する．直交格子を用いた複数 GPU 計算では，一般に領域分割法を用い並列化する．図 1 は計算格子の領域分割を表している．

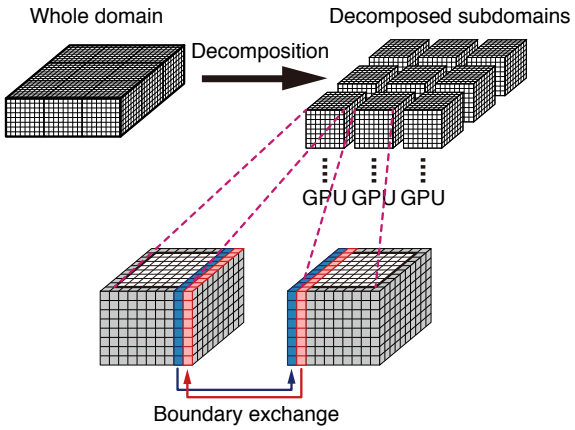

図 1 直交格子計算の複数 GPU による並列化
Fig. 1 Multi-GPU computing of mesh-based computation.

本フレームワークでは，分割された計算格子それぞれに一つの GPU が割り当てられる．領域分割法では，各計算格子の境界領域がステンシル計算で必要となるため，隣接領域間で境界領域を交換しながら計算を進める（図 1）．

本フレームワークでは，同一ノード内の GPU 間通信を効率的に行うため，計算に用いられる全 GPU の並列化を MPI で行わず，ノード内の複数 GPU は単一プロセスで扱い，ノード間を MPI で並列化する．ユーザコードでは単一プロセスが複数 GPU を扱うことを意識しないよう各プロセス内では OpenMP によるスレッドで並列化し，ノード内の複数 GPU を扱う．一つのスレッドに一つの GPU を割り当てる．図 2 に本フレームワークの MPI と OpenMP による複数 GPU の扱いについて示す．ユーザコードの MPI と OpenMP による並列化は本フレームワークによって実行され，ユーザは図 2 に示された赤枠内に着目して，OpenMP のスレッド上で物理変数のための配列を確保し，それを用いて時間発展計算を記述することになる．

3.2 ステンシル計算関数の実行

本フレームワークでは，OpenMP のスレッド内で時間発展計算が実行される．OpenMP スレッド内で 1 つの GPU を用い格子上の計算を行う．

本フレームワークでは，ユーザはある格子点を更新するステンシル計算関数を記述する．本フレームワークは，このステンシル計算関数を全格子点に適用するための C++用のクラスを提供している．このクラスは，ステンシル計算関数を関数オブジェクトとして受け取り，CUDA のグローバル関数として実行する．3 次元計算で計算格子サイズが (n_x, n_y, n_z) であるとき，CUDA block は $(64, 2, n_z/16)$ と確保し，CUDA thread 内で z 方向に 16 格子点マーチングしながら計算を行う．現在，フレームワークは，ステンシル計算関数を GPU の他に CPU でも実行可能である．ステンシル計算関数にポインタ型の引数が渡されると，フレームワークはそのポインタが GPU 上のメモリかホスト

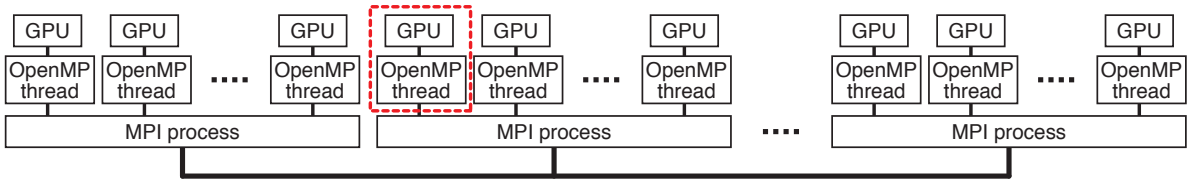

図 2 MPI と OpenMP を用いた複数 GPU 計算
Fig. 2 Multi-GPU computing by using both MPI and OpenMP.

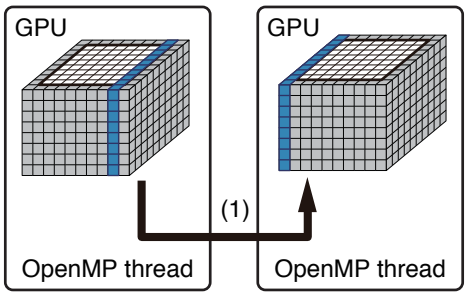

図 3 同一ノード内の OpenMP による GPU 間通信
Fig. 3 Intra-node GPU-GPU communication by the OpenMP threads.

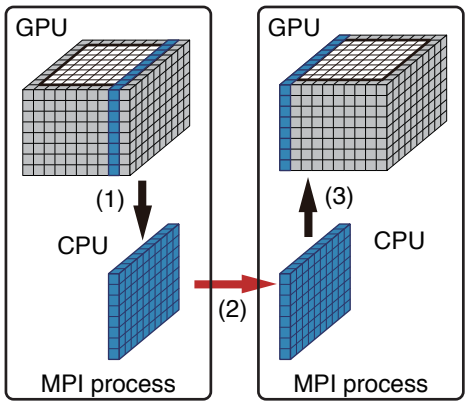

図 4 異なるノード間の MPI による GPU 間通信
Fig. 4 Inter-node GPU-GPU communication by MPI.

上のメモリを指すか判定し，GPU または CPU の適したデバイスでステンシル計算関数を実行する．CPU 上では，ステンシル計算関数は `for` 文で実行される．

3.3 同一ノード内の GPU 間通信

本フレームワークでは，同一ノード内の複数 GPU 計算は OpenMP の複数スレッドが担当する．一つのスレッドが一つの GPU を担当する．ノード内の全スレッドが異なるスレッド（すなわち異なる GPU）で確保された配列にアクセスできるよう通信を行う配列のポインタは通信前にフレームワーク内に登録される．登録されたポインタを参照することで，OpenMP スレッドは異なるスレッドで確保された配列にアクセスできる．同一ノード内の GPU 間通信は，CUDA API の `cudaMemcpy` で異なる GPU 上に確保された配列へのポインタを指定し実現している．特に通信を行う 2 つの GPU が GPUDirect による peer-to-peer 通信に対応している場合は，図 3 に示すように送信元のデバイスメモリを直接参照でき，通常の複数 GPU 計算で用いられるフラット MPI による実装と比べ，より高速な通信が行える．

3.4 異なるノードの GPU 間通信

ノード間は MPI により並列化されている．異なるノード間の GPU 間通信は通信相手の GPU メモリを直接参照することはできないため，図 4 に示すように
(1) GPU メモリからホストメモリへのデータコピー
(2) MPI によるホストメモリの送受信
(3) ホストメモリから GPU メモリへのデータコピー
の 3 段階で実行している．MPI 通信に必要なホスト上のバッファはフレームワークが自動的に確保する．

4. プログラミングモデル

本フレームワークは，C++言語から利用できる．フレームワークは C++用テンプレートクラスにより次の機能を提供する．

- 複数スレッドによる並列実行を行うためのクラス
- 各スレッドが担当する計算格子サイズを取得する関数
- ステンシル計算関数を記述するためのステンシルアクセスを表現するクラス
- ステンシル計算関数を実行するためのクラス
- 配列変数の GPU 間通信を行うクラス

ユーザはこれらの機能を用い，(1) ある GPU に着目して物理変数を保持する配列を確保し (2) その物理変数を更新するステンシル計算を行う関数をプログラムする．(3) GPU 間通信用クラスを用い，物理変数の境界領域の交換を行うことになる．本章では拡散方程式を例にとってプログラミングモデルを説明する．

4.1 複数プロセスおよび複数スレッドによる並列実行

本フレームワークでは，ユーザプログラムの開始時にフレームワークの提供する C++クラス `DomainGroup` で計算領域と複数スレッドによる並列実行環境を生成する．ユーザプログラムは MPI として実行し，MPI の各プロセスで次のコードを実行する．

```
DomainManager manager(px, py, pz);
DomainSize domsize(nx, ny, nz, mgnx, mgny, mgnz);
manager->
  init_domain_size_by_local_domain_size(domsize);
```

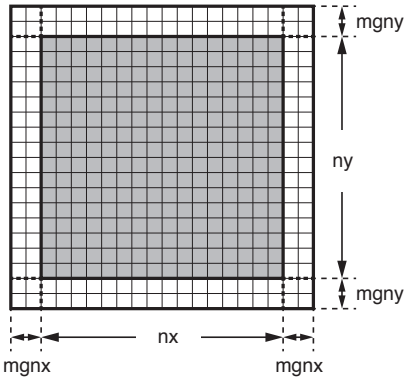

図 5 1 つの GPU が計算を行う計算領域の xy 断面
Fig. 5 X-Y plance of a computational subdomain that assigned to a GPU.

```
manager->set_thread_assignment(nthreads);
DomainGroup domain_group(rank, &manager);
domain_group.run(main_run);
```

まず DomainManager を使い，計算領域の 3 次元の分割数 (px，py，pz) を指定する．DomainSize は計算格子サイズを指定するもので，ここでは計算領域に (nx，ny，nz) の 3 次元格子を指定し，境界領域として各方向に mgnx，mgny，mgnz 個の格子を持つ領域を指定している．この格子サイズを各スレッド，すなわち各 GPU が計算する領域のサイズとして設定している．図 5 に各 GPU が計算する領域サイズの xy 断面を示す．図中のグレーの領域を GPU で計算し，白色の領域は隣接 GPU から送信されたデータを格納する境界領域である．各 MPI プロセス内で生成するスレッド数 nthreads を DomainManager に渡し，この DomainManager と MPI のランク番号 rank を使い，DomainGroup のオブジェクトを生成する．DomainGroup は内部で OpenMP の並列リージョンで nthreads 数のスレッドを生成し，そのスレッド内で run() 関数で指定された関数 main_run() を実行する．

関数 main_run() はユーザ定義関数で，この関数内で変数配列の確保，時間発展計算を行う．main_run() は一般に以下のようなコードとなる．

```
int main_run(const Domain &domain) {
  const DomainSize &domsize
      = domain.local_domain_size();
  float *f, *fn;
  cudaMalloc(&f , domsize.ln()*sizeof(float));
  cudaMalloc(&fn, domsize.ln()*sizeof(float));
  initialize_diffusion(domsize, f);
  ...
}
```

DomainGroup::run() で指定された関数は Domain を受け取る．Domain は各 OpenMP スレッドが担当する領域の情報を保持し，Domain.local_domain_size() によりスレッドの担当する計算領域サイズを保持するクラス DomainSize を取得できる．ユーザは DomainSize を用い，物理変数を保持する配列 f，fn を C++言語の通常の配列とし確保し，これらの変数を初期化することになる．DomainSize::ln() は (nx+2*mgnx) * (ny+2*mgny) * (nz+2*mgnz) を返す関数である．

4.2 ステンシルアクセスの表現

ユーザ関数 main_run() 内で物理変数の時間更新するステンシル計算を行う関数を実行する．本フレームワークは，ステンシルアクセスを表現するため，インデックスを記述するクラス ArrayIndex3D （3 次元計算用）等を提供する．これを用い，ステンシル計算関数を記述する．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，ArrayIndex3D.ix() は f[ArrayIndex3D.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，ArrayIndex3D.ix<+1，0，0>()，ArrayIndex3D.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，インデックス計算の高速化を図っている．

4.3 ステンシル計算関数の定義と実行

ステンシル計算関数は，ArrayIndex3D 等を用い，ファンクタ（関数オブジェクト）として定義する．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>()];
}
};
```

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．

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を以下のように実行する．

```
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);
```

とする．Loop3D::run() は任意個の任意の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．このためユーザは自由にファンクタを定義できる．ファンクタは，__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．フレームワークは，CPU と GPU それぞれに特化した最適化コードは記述できず，同一のファンクタを実行することになる．

Loop3D のコンストラクタは，第一，第二，第三引数が n_x，i_0，i_1 であるとき，x 方向の格子点数を n_x とし x 方向にインデックス i_0 から $n_x - i_1 - 1$ までステンシル計算関数を実行する．上の例では，x 方向の格子点数は nx+2*mgnx となり，Diffusion3d() は x 方向に対して mgnx から nx+mgnx-1 までの格子点に対して実行される．Diffusion3d() は境界領域では実行されないこととなる．第四引数から第九引数で y，z 方向について x 方向と同様に指定する．

Loop3D 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，適切な CUDA の block 数，thread 数を渡し全格子点に対して実行する．第二引数以降で初めて出てくるポインタが GPU メモリを指すか CPU メモリを指すかを判定し，GPU で実行するか CPU で実行するかを自動的に決定する．

4.4 変数の GPU 間通信と境界条件の設定

本フレームワークは，複数 GPU 計算に対応するため，GPU 間通信を簡単に記述するためのクラス BoundaryExchange を提供する．複数 GPU における通信を効率的に行うため，ノード内並列では OpenMP を用い，ノード間並列では MPI を利用する．また，ノード内で可能な場合は GPUDirect を利用している．

クラス BoundaryExchange を用いて，以下のように GPU 間通信を行う．

```
BoundaryExchange *exchange = domain.exchange();
exchange->append(f);
exchange->transfer();
```

domain はクラス Domain のオブジェクトである．クラス BoundaryExchange は，Domain で初期化され格子サイズ等が設定され隣接 GPU への転送量を把握する．転送する配列は，BoundaryExchange::append() で登録する．ここでは変数 f を登録している．BoundaryExchange::append() で任意の数の配列を登録することが可能である．登録された配列は，一時的に BoundaryExchange に格納される．BoundaryExchange::transfer() が実行されると MPI 通信が必要な境界領域のデータは，ホストメモリのバッファを経由し隣接ノードへ送信される．BoundaryExchange は OpenMP の全スレッドで共有されているため，登録されたポインタを直接参照することでノード内の GPU へ送信する．

境界条件のうち周期境界条件は GPU 間の通信の設定と同時に BoundaryExchange を用い設定される．周期境界条件以外の境界条件は，BoundaryExchange と同じように使えるフレームワークが提供するクラス BoundaryCondition を用いて設定する．このクラスではディリクレ条件，ノイマン条件の境界条件を設定できる．これよりも複雑な境界条件では，境界にのみ適用されるステンシル計算関数を定義する必要がある．

5. 評価実験

提案フレームワークの有用性を示すために，まず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．次に，本フレームワークを実アプリケーションへ適用できることを示す．フレームワークを用い拡散方程式と比較し複雑な Euler 方程式を実装し，Rayleigh-Taylor 不安定性計算を行う．

5.1 拡散方程式による性能評価

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．

$$\frac{\partial f}{\partial t} = \kappa \nabla^2 f \qquad (1)$$

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．隣接 GPU から送信されるデータを保持する境界領域は，1 格子点の厚さが必要となる．

性能測定は，東京工業大学の GPU スパコン TSUBAME2.5 を用いる．TSUBAME2.5 は 4000 基を超える NVIDIA Tesla K20X GPU が搭載されている．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．TSUBAME2.5 の 1 ノードには Intel CPU Xeon X5670 (Westmere-EP) 2.93 GHz 6-core が 2 ソケット，Tesla K20X が 3 基搭載されている．

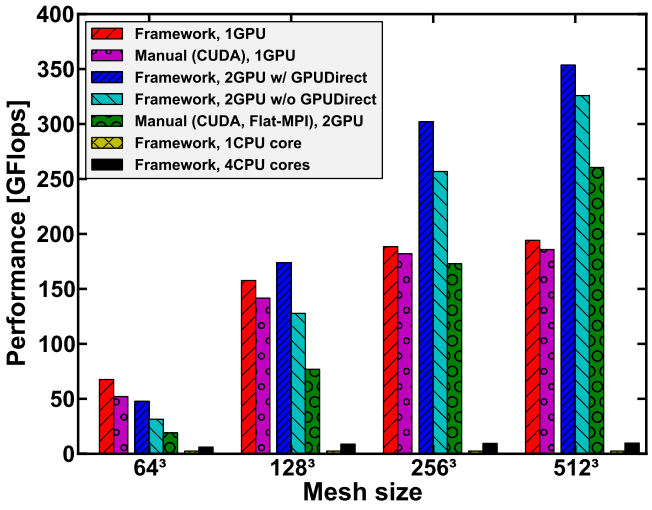

図 6 フレームワークおよび手による実装による拡散計算の実行性能比較

Fig. 6 Performance of diffusion computation obtained by the proposed framework and manual implementation.

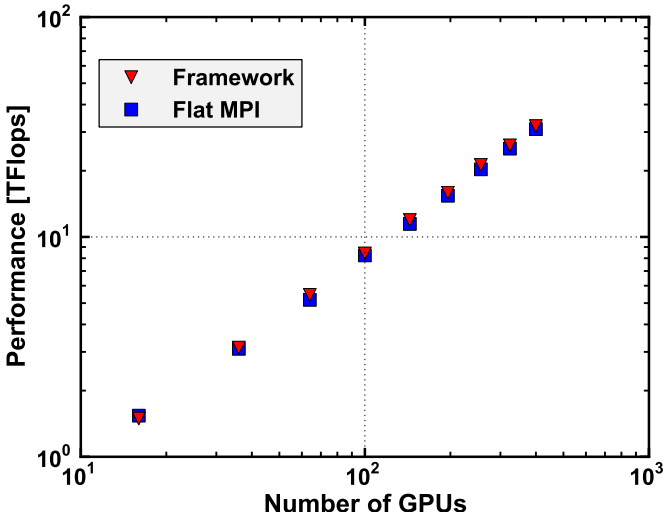

図 7 TSUBAME2.5 による拡散計算の弱スケーリング

Fig. 7 Weak scaling results of diffusion computation on TSUBAME2.5.

図 6 に本フレームワークおよび手による実装による拡散計算の実行性能を示す．格子サイズを 64^3 から 512^3 まで変化させ，単一ノード内の 1GPU あるいは 2GPU を使用して拡散方程式を実行する．1GPU および 2GPU とも，フレームワークを使用した場合とフレームワークを使用せず手による実装をした場合の性能を示している．手による実装では 2GPU 間を MPI により並列化している．2GPU の場合，フレームワーク上で使用する 2GPU 間で GPUDirect により直接通信を有効とした場合と無効とした場合の性能を示している．参照として，フレームワークは，CPU を用いて実行することも可能であるため，1 CPU コアおよび 4CPU コアを使用した場合の性能を合わせて示す．

図に示すように，1GPU 計算においてフレームワークは手による実装よりも高性能を達している．512^3 ではフレームワークは 194.2 GFlops に達し，これは手による実装の 1.04 倍である．2GPU 計算では，MPI により並列化された手による実装と比べてフレームワークを用いた計算は高い実行性能を達成している．特に 2 つの GPU で直接通信が可能な GPUDirect を有効とした場合，高い性能を示し，512^3 では 353.7 GFlops に達する．これは 1GPU による性能の 1.82 倍で，GPU 間を直接アクセスする高速な通信のため性能低下の割合が少ない．一方，MPI による並列化では GPU 間のデータ通信は必ずホストメモリを経由することとなり，性能が大幅に低下する．MPI による実装と比較しフレームワークによる 353.7 GFlops の性能は，1.4 倍高速である．フレームワークは CPU 上で実行することも可能で，GPU による性能と比較すると低いものの，512^3 では 4CPU コアで 9.7 GFlops に達し，これは 1CPU コアの性能の 3.9 倍である．

次に図 7 に TSUBAME2.5 を用いた拡散計算の弱スケーリングの結果を示す．フレームワークを用いた実装と，計算に使用する全 GPU を MPI で並列化（フラット MPI）する手による実装との性能を比較する．ともに 1 ノードあたり 3GPU を使用する．ただし，フレームワークでは，1 ノードに 1MPI ランクを割り当て 3 スレッドにより 3GPU を制御する．一方，手による実装では，ノード内の各 GPU に 1MPI ランクを割り当て，1 ノードで 3 プロセスを実行する．計算格子は 1GPU あたり $1024 \times 256 \times 256$ とする．図に示すように，フレームワークによる実装は手による実装とほぼ同等かそれ以上の実行性能を達成し，400GPU の性能は 16GPU の性能と比較し 85.6% で良いスケーリングを達成している．

5.2 流体計算への適用例

本フレームワークの実問題への適用例とし，圧縮性流体計算として 3 次元 Euler 方程式をフレームワークを用い実装し，Rayleigh-Taylor 不安定性の成長シミュレーションを行う．次の方程式を解く．

$$\frac{\partial U}{\partial t} + \frac{\partial E}{\partial x} + \frac{\partial F}{\partial y} + \frac{\partial G}{\partial z} = S, \tag{2}$$

$$U = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e \end{bmatrix}, \; E = \begin{bmatrix} \rho u \\ \rho uu + p \\ \rho vu \\ \rho wu \\ (\rho e + p)u \end{bmatrix}, \; F = \begin{bmatrix} \rho v \\ \rho uv \\ \rho vv + p \\ \rho wv \\ (\rho e + p)v \end{bmatrix},$$

$$G = \begin{bmatrix} \rho w \\ \rho uw \\ \rho vw \\ \rho ww + p \\ (\rho e + p)w \end{bmatrix}, \; S = \begin{bmatrix} 0 \\ 0 \\ 0 \\ \rho g \\ \rho wg \end{bmatrix}$$

ここで，ρ は密度，(u, v, w) は速度，p は圧力，e はエネルギーを表している．g は重力加速度である．移流計算は保存型の 3 次精度風上手法で解き，時間積分は低メモリ消費

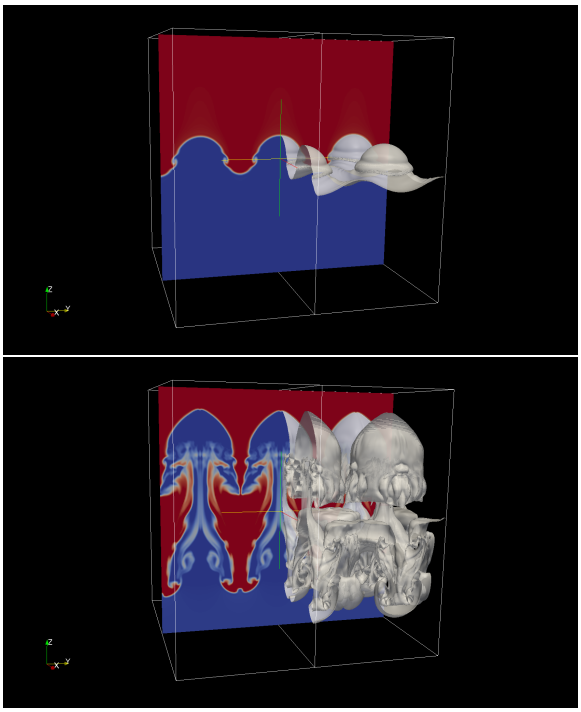

図 8 Rayleigh-Taylor 不安定性シミュレーション
Fig. 8 Simulation results of the Rayleigh-Taylor instability.

型の 3 段 3 次精度の TVD Runge-Kutta 法を用いる．

拡散計算では扱う変数が f のみであるが，本計算では $\rho, \rho u, \rho v, \rho w, \rho e$ の 5 変数の時間発展を解く．ステンシルは 1 方向に 5 点，3 次元計算では 13 点の格子点を参照する．隣接 GPU から送信されるデータを保持する境界領域は 2 格子点の厚さが必要となる．フレームワークを用いることで，通信部分は以下のように簡便に記述可能となる．

```
BoundaryExchange *exchange = domain.exchange();
exchange->append(vars->r);
exchange->append(vars->ru);
exchange->append(vars->rv);
exchange->append(vars->rw);
exchange->append(vars->re);
exchange->transfer();
```

vars は上記の 5 変数を保持する構造体である．

図 8 にフレームワークを用い実装した圧縮性流体計算で得られた計算結果例を示す．TSUBAME2.5 の 12 GPU を用い計算した．赤色と青色は yz 平面の密度を表しており，界面は密度が変化する領域を示している．界面は一部分のみ可視化した．フレームワークを利用することで実アプリケーションを簡便に実装でき，提案フレームワークは複雑な計算に対しても適用可能である．

6. おわりに

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マルチ GPU コンピューティング・フレームワークを提案した．従来の GPU 用コード開発を支援するフレームワークや DSL とは異なり，言語拡張や標準でないプログラミングモデルを導入することなく，通常の C++コードを記述することで GPU スパコン向けの最適化を施すことが可能である．ユーザコードは C++言語で記述できるため可搬性と拡張性が高い．提案フレームワークの実装には，移植性を考慮して広く使われる C++言語と CUDA を用いている．

提案フレームワークは，同一ノード内の GPU 間通信を効率的に行うため，ノード内の複数 GPU を OpenMP のスレッドで扱い，ノード間を MPI で並列化する．ユーザは一つの OpenMP スレッド上で，物理変数のための配列を確保し，それを用いて時間発展計算を記述する．物理変数の時間更新を可搬性高く簡便に記述するため，格子点上のステンシル計算を表現するクラスとそのステンシル計算を全格子に渡って実行するクラスを提供する．複数 GPU 計算を行うため，GPU 間通信を簡単に行うクラスを提供する．ノード内の GPU 間通信では，できる限り直接 GPU メモリを参照するよう実装されている．ノード間の GPU 間通信はホストメモリを経由する MPI 通信を行う．

評価実験では，提案フレームワークを用い実装した複数 GPU の拡散計算を東京工業大学の TSUBAME2.5 で実行し，手による実装と比較して高い性能を達成することを示した．ノード内の 2GPU を用いた計算では，MPI を用いた実装と比較して 1.4 倍の高速化に成功した．提案フレームワークの実問題への適用例として，圧縮性流体計算を本フレームワークを用い実装し，複雑な計算に対しても適用可能であることを示した．

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（B）課題番号 23360046 「GPU スパコンによる気液二相流と物体の相互作用の超大規模シミュレーション」，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] 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] Michalakes, J. and Vachharajani, M.: GPU acceleration of numerical weather prediction., *IPDPS*, IEEE, pp. 1–7 (2008).

[5] Linford, J. C., Michalakes, J., Vachharajani, M. and Sandu, A.: Multi-core acceleration of chemical kinetics for simulation and prediction, *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, New York, NY, USA, ACM, pp. 1–11 (online), DOI: http://doi.acm.org/10.1145/1654059.1654067 (2009).

[6] Hamada, T. and Nitadori, K.: 190 TFlops Astrophysical N-body Simulation on a Cluster of GPUs, *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. 1–9 (online), DOI: http://dx.doi.org/10.1109/SC.2010.1 (2010).

[7] Feichtinger, C., Habich, J., Köstler, H., Hager, G., Rüde, U. and Wellein, G.: A Flexible Patch-Based Lattice Boltzmann Parallelization Approach for Heterogeneous GPU–CPU Clusters, *Parallel Computing*, Vol. 37, No. 9, pp. 536–549 (2011).

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

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

[10] 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).

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